Chapter 15: Design Examples

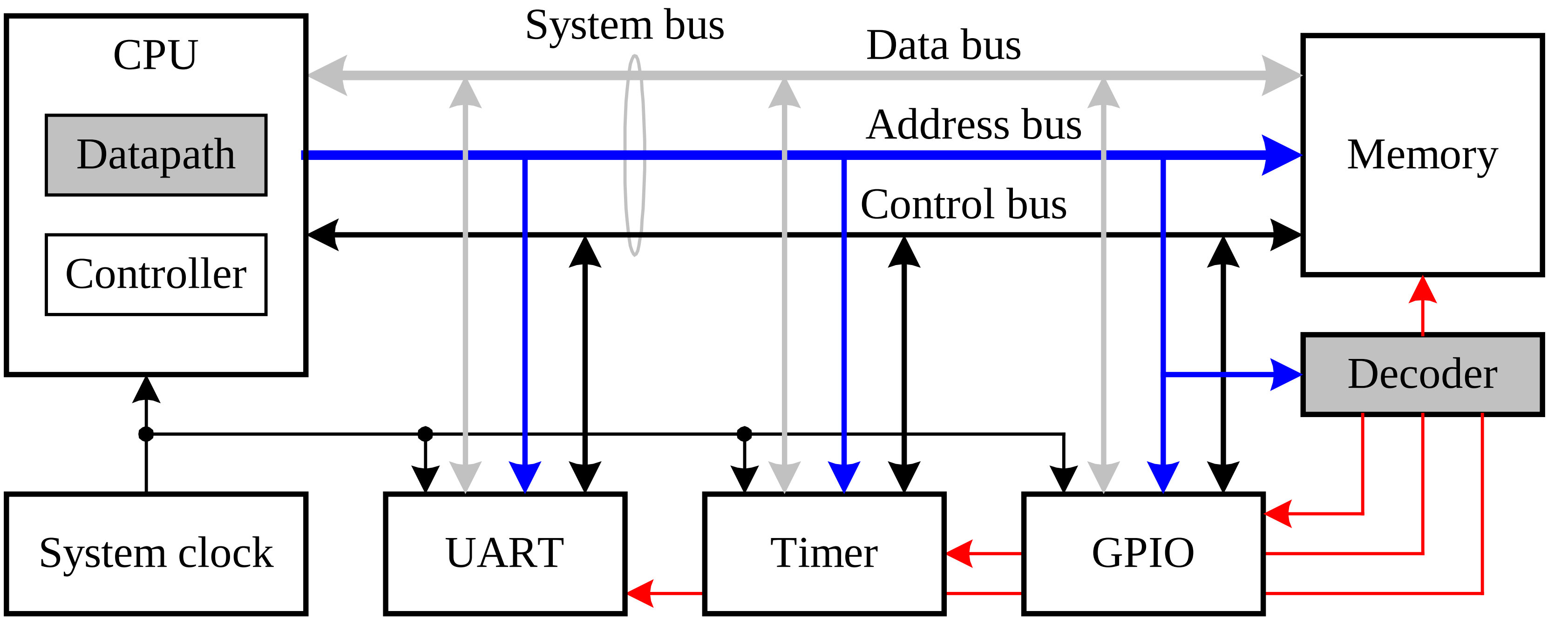
Prof. Soo-Ik Chae

Objectives

After completing this chapter, you will be able to:

- ❖ Describe basic structures of μP systems
- ❖ Understand the basic operations of bus structures
- ❖ Understand the essential operations of data transfer
- ❖ Understand the design principles of GPIOs
- ❖ Understand the design principles of timers
- ❖ Understand the design principles of UARTs
- ❖ Describe the design principles of CPUs

A Basic μP System

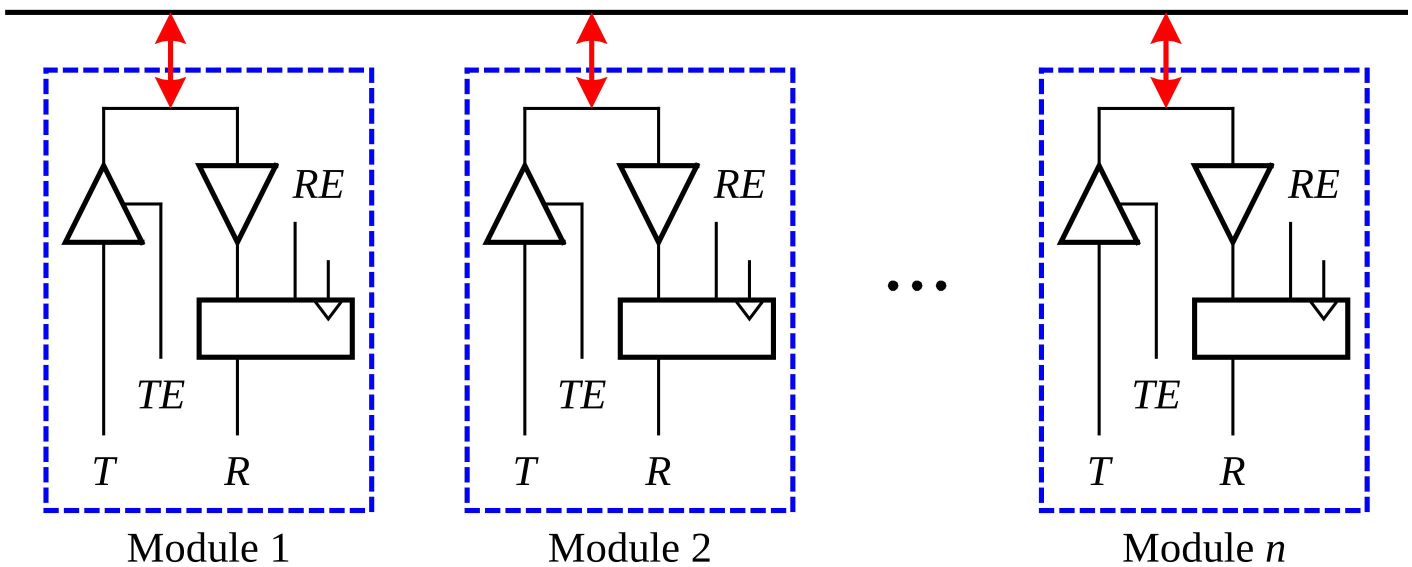


Bus Structures

- A bus is a set of wires used to transport information between two or more devices in a digital system.
- Types of buses:
  - tristate bus
    - When realizing by using tristate buffers.
  - multiplexer-based bus
    - When realizing by using multiplexers.
- The tristate bus is often called bus for short.

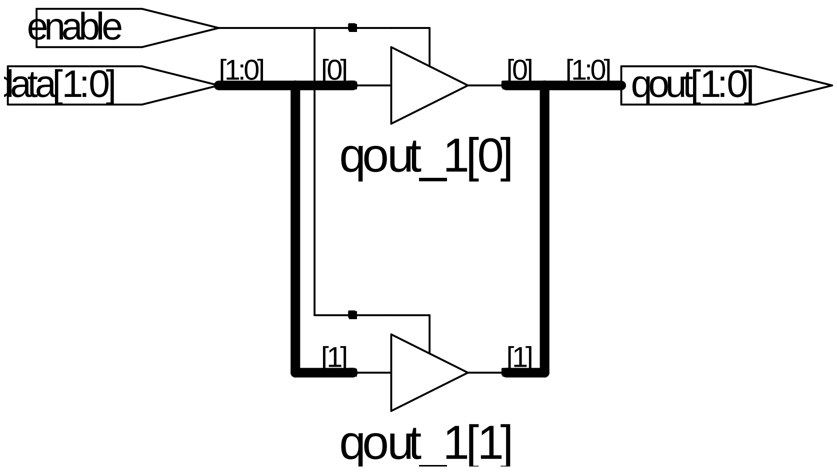
A Tristate Bus

- Tristate bus
  - A bidirectional interface drives a signal *T* to the bus and samples a signal on the bus onto an internal signal *R*.
  - Only one module is allowed to transmit signal on the bus.

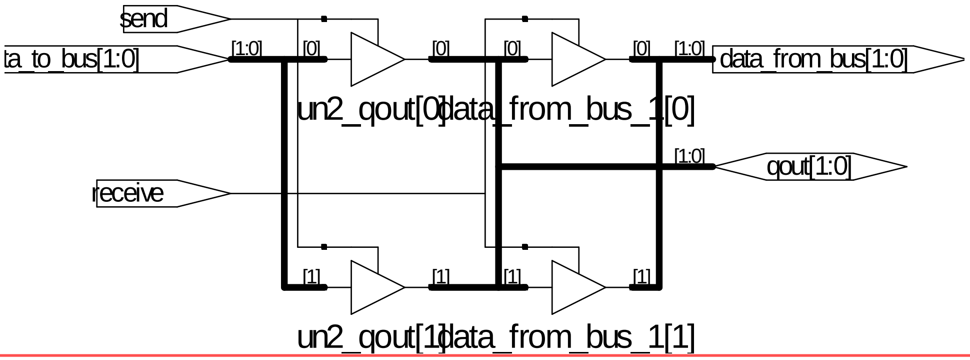


A Tristate Bus Example

```
// a tristate bus example
module tristate_bus (data, enable, qout);
parameter N = 2;   // define bus width
input   enable;
input   [N-1:0] data;
output [N-1:0] qout;
wire    [N-1:0] qout;

// the body of tristate bus
assign qout = enable ? data : {N{1'bz}};
endmodule
```



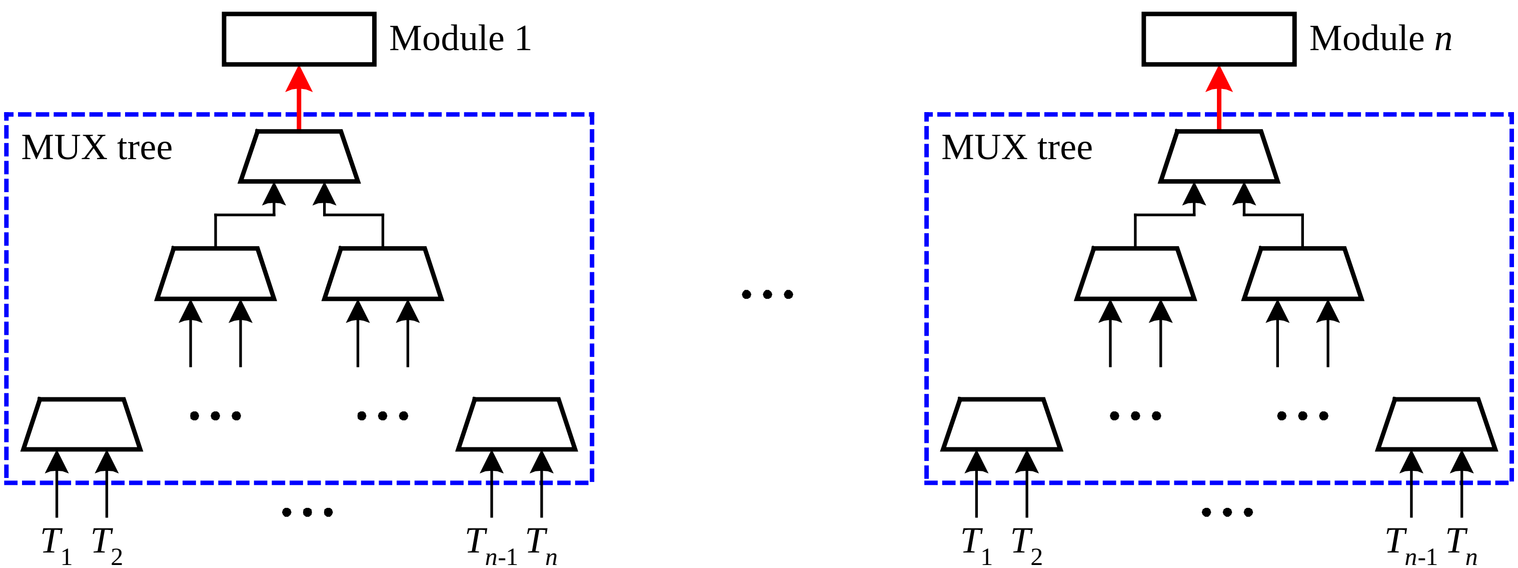
A Bidirectional Bus Example




```
// a bidirectional bus example
module bidirectional_bus (data_to_bus, send, receive, data_from_bus, qout);
parameter N = 2;              // define bus width
input  send, receive;
input  [N-1:0] data_to_bus;
output [N-1:0] data_from_bus;
inout  [N-1:0] qout;          // bidirectional bus
wire   [N-1:0] qout, data_from_bus;
// the body of tristate bus
assign data_from_bus = receive ? qout : {N{1'bz}};
assign          qout = send ? data_to_bus : {N{1'bz}};
endmodule
```

A Multiplexer-Based Bus

- Multiplexer-based bus
  - It can avoid the large amount of capacitive load.
  - It has much less the propagation delay than the tristate bus when the number of modules attached to it is large enough.

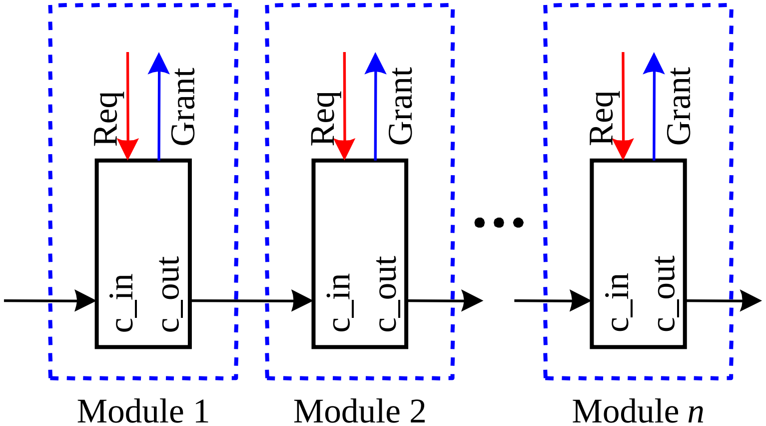


Bus Arbitration

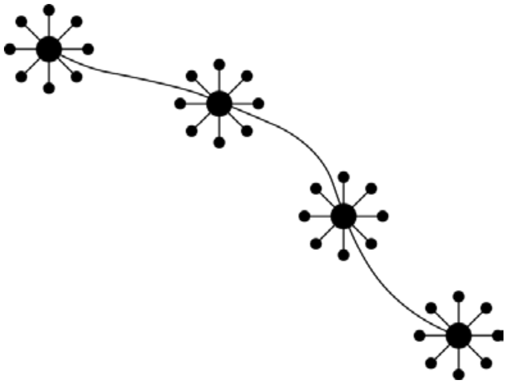
- The operation that chooses one transmitter from multiple ones attempting to transmits data on the bus is called a bus arbitration.
- The device used to perform the function of bus arbitration is known as a bus arbiter.

Daisy-Chain Arbitration

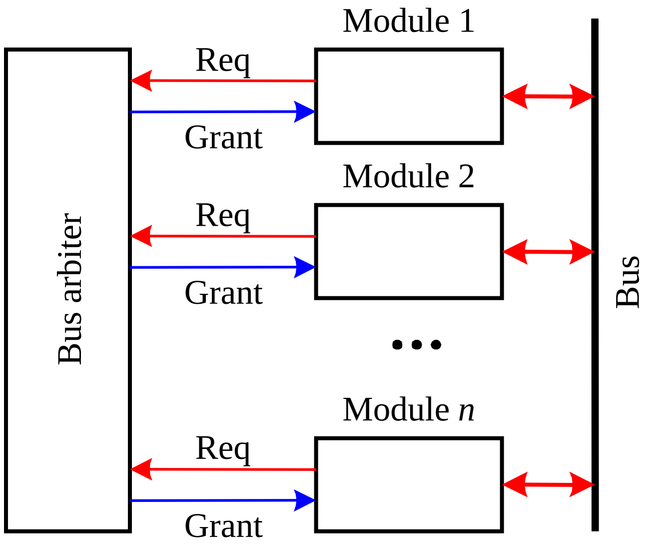
- 소자가 직렬로 연결되고 신호가 한 소자에서 다른 소자로 통과하는 버스를 따라 신호를 전달하는 방식. 데이지 체인(daisy chain) 체계는 버스 상의 소자의 전기적 위치에 기반하여 소자의 우선 순위를 할당한다.



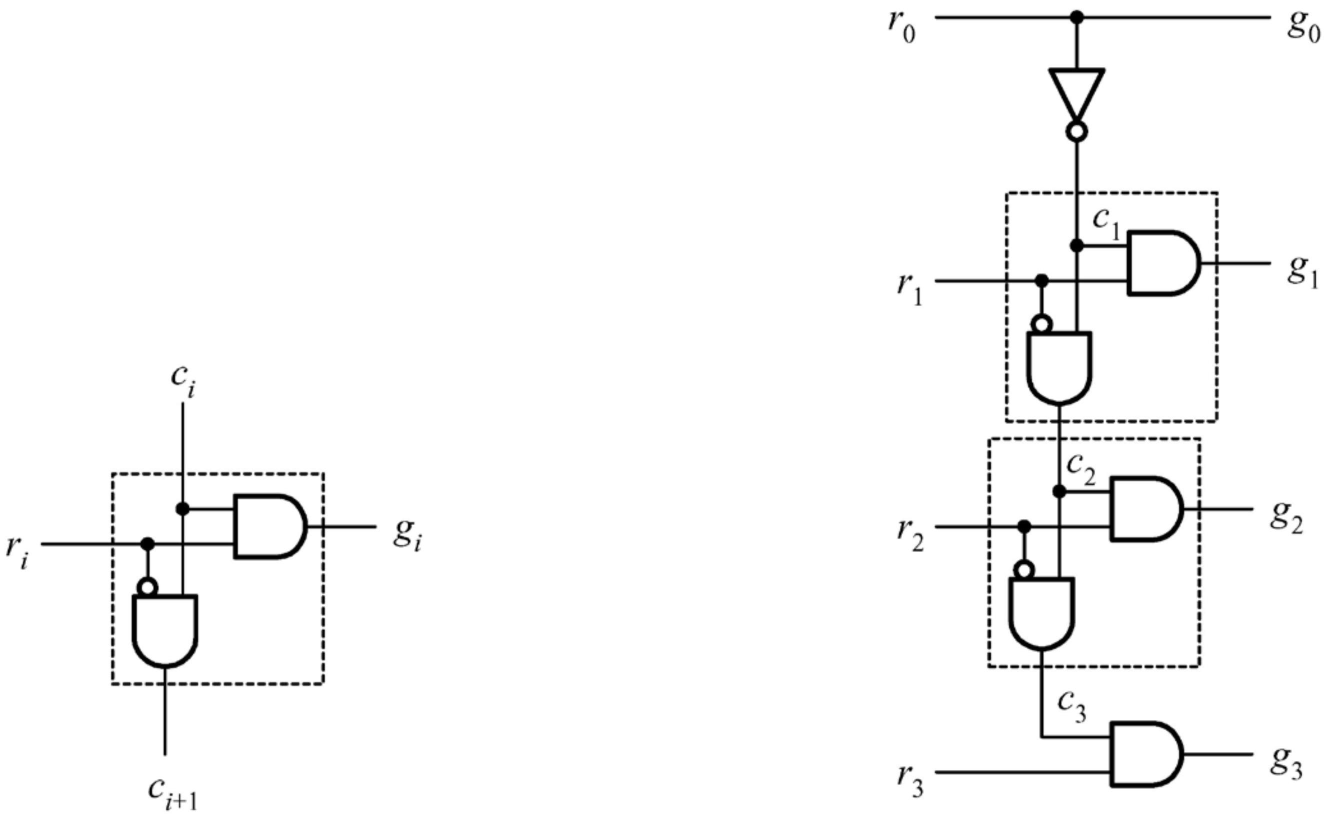
(a) Daisy-chain arbitration



Centralized arbitration



Daisy-Chain Arbitration

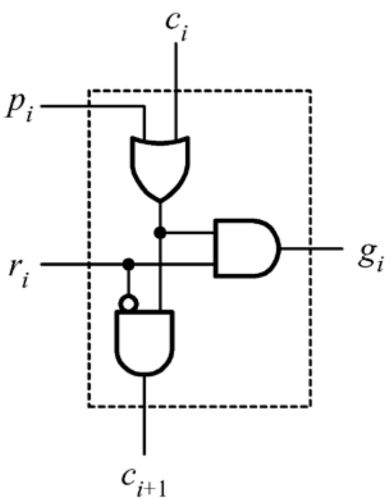


(a) A bit cell

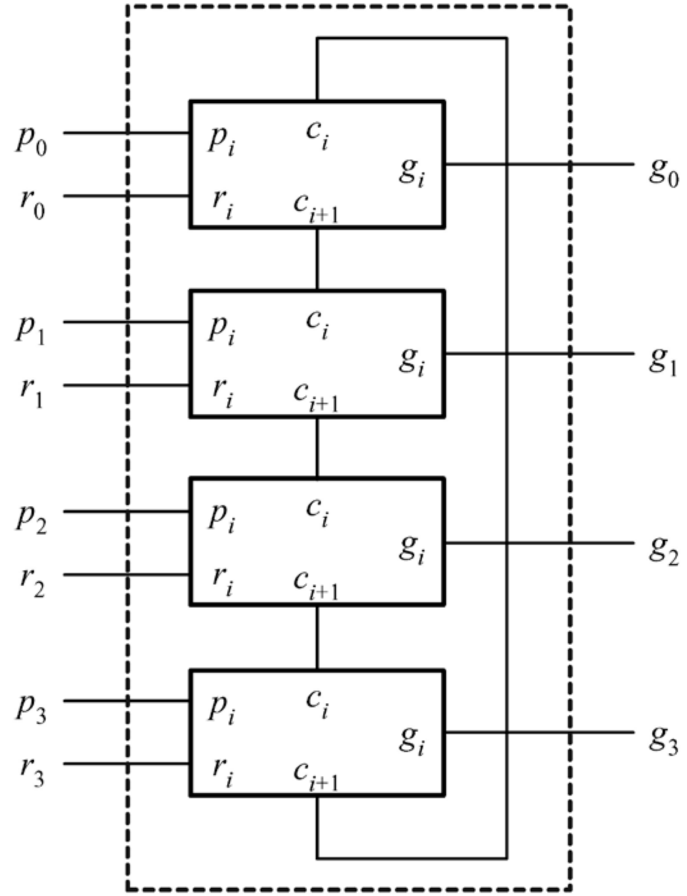
(b) A 4-request daisy-chain arbiter

Figure 15.6: The logic diagram of a 4-request daisy-chain bus arbiter.

Daisy-Chain Arbitration



(a) A bit cell



(b) A 4-request variable-priority arbiter

Figure 15.7: The logic diagram of a 4-request variable-priority bus arbiter.

Round-Robin Arbitration

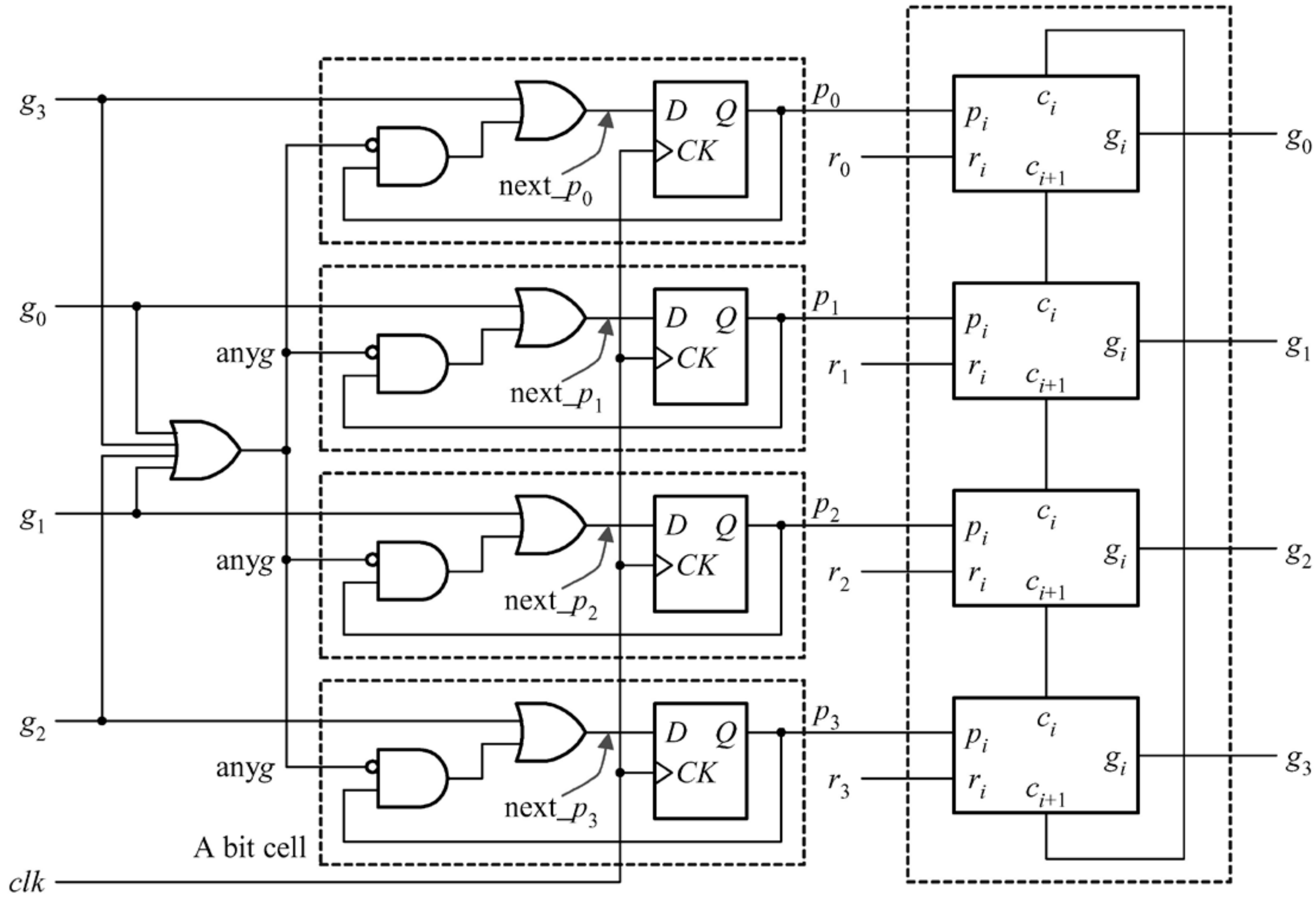


Figure 15.8: The logic diagram of a 4-request round-robin bus arbiter.

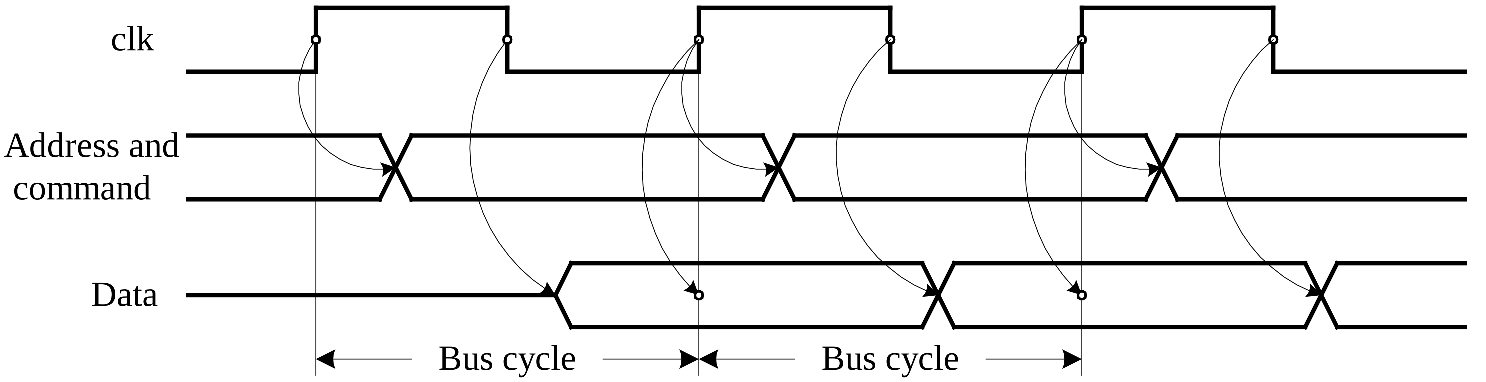
Data Transfer Modes

- Data transfer modes:
  - synchronous mode
  - asynchronous mode
- Regardless of data transfer modes, the actual data can be transferred in:
  - parallel: a bundle of signals in parallel
  - serial: a stream of bits

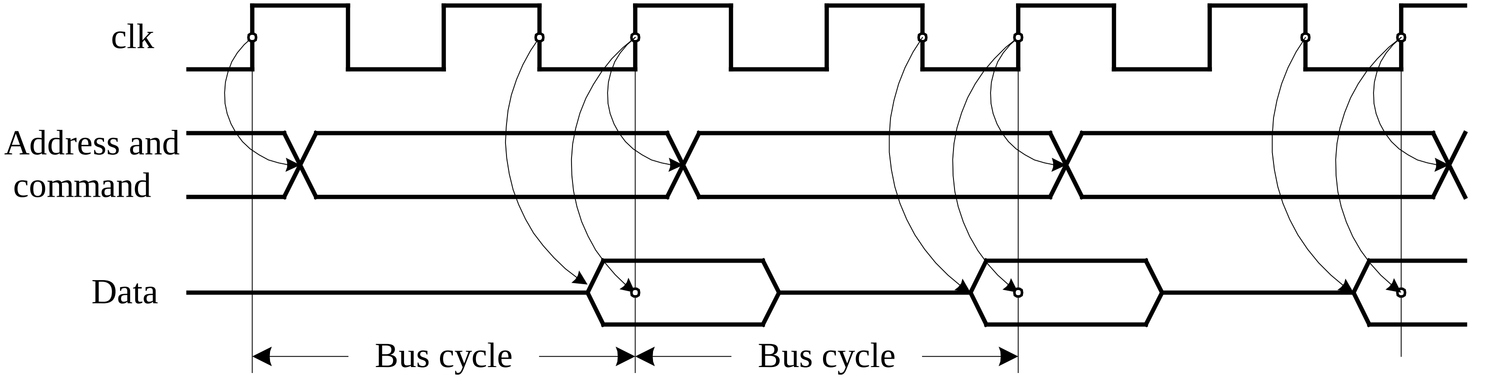
Synchronously Parallel Data Transfers

- Each data transfer is in synchronism with clock signal.
  - Bus master: A device generates address and command.
  - Bus slave: A device receives the address and the command from the bus.
- Synchronous bus transfers can be further divided into two types:
  - Single-clock bus cycle
  - Multiple-clock bus cycle

Synchronously Parallel Data Transfers



(a) Single-clock bus cycle



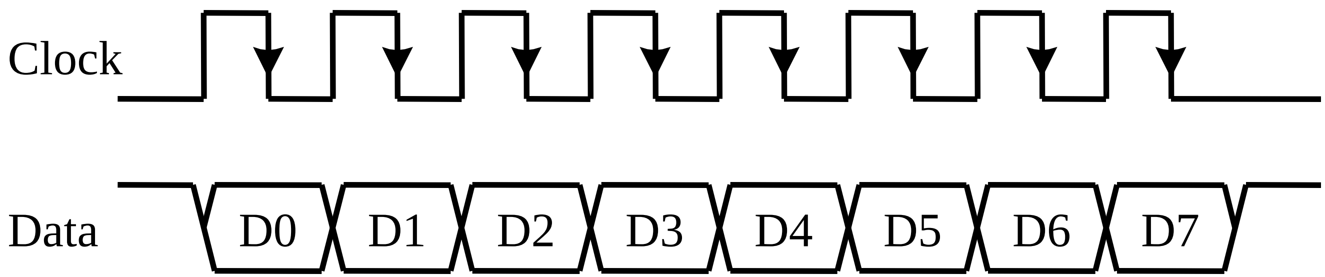
(b) Multiple-clock bus cycle

Synchronously Serial Data Transfers

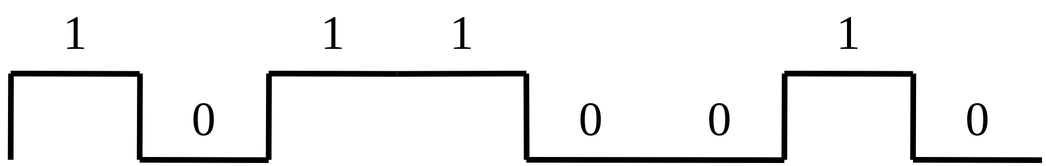
- In synchronous serial data transfer, the clock signal is sending along with the data.
  - Explicitly clocking scheme
    - The clock signal is sent along with data explicitly as a separate signal.
  - Implicitly clocking scheme
    - The clock signal is encoded into the data stream.
    - The clock signal is then extracted at the receiver before sampling the data.

Synchronously Serial Data Transfers

❖ Examples of synchronously serial data transfer



(a) Serial data transfer with explicitly clocking



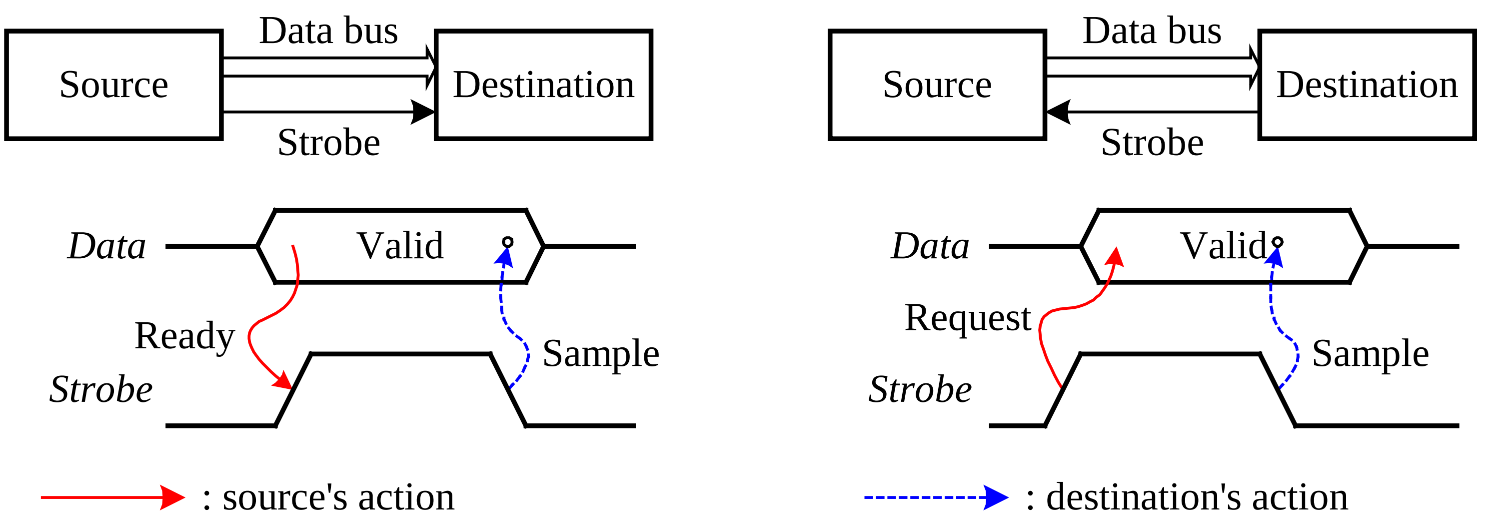
(b) Serial data transfer with implicitly clocking (NRZ code)

Asynchronous Data Transfers

- Each data transfer occurs at random
- The data transfer may be controlled by using:
  - strobe scheme
  - handshaking scheme
- Both strobe and handshaking are used extensively on numerous occasions that require the transfer of data between two asynchronous devices.

Strobe

- Only one control signal known as strobe is needed.
- The strobe signal is enabled by either the source device or destination device, depending on the actual application.



(a) Source-initiated transfer

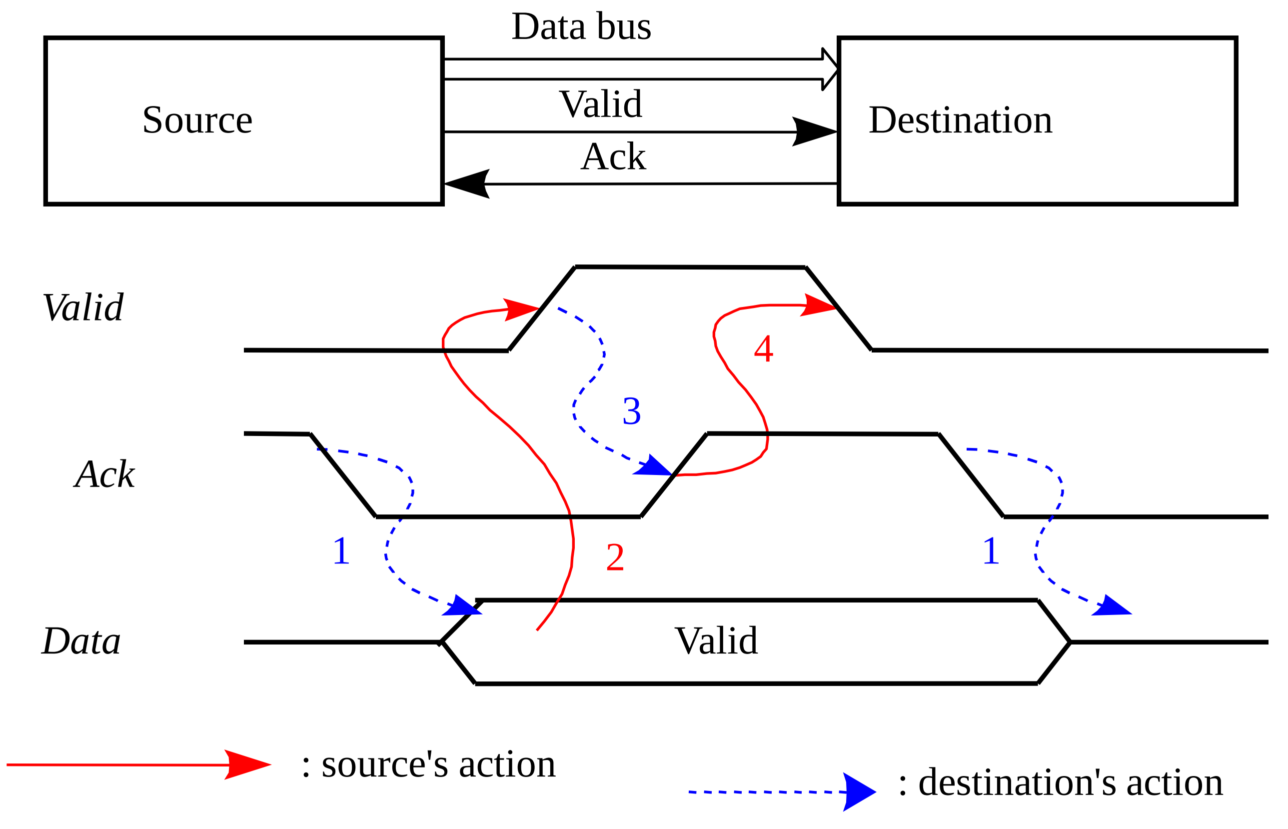
(b) Destination-initiated transfer

Handshaking

- In the handshaking transfer, four events are proceeded in a cycle order:
  1. ready (request):
  2. data valid:
  3. data acceptance:
  4. acknowledge:

Source-initiated transfer

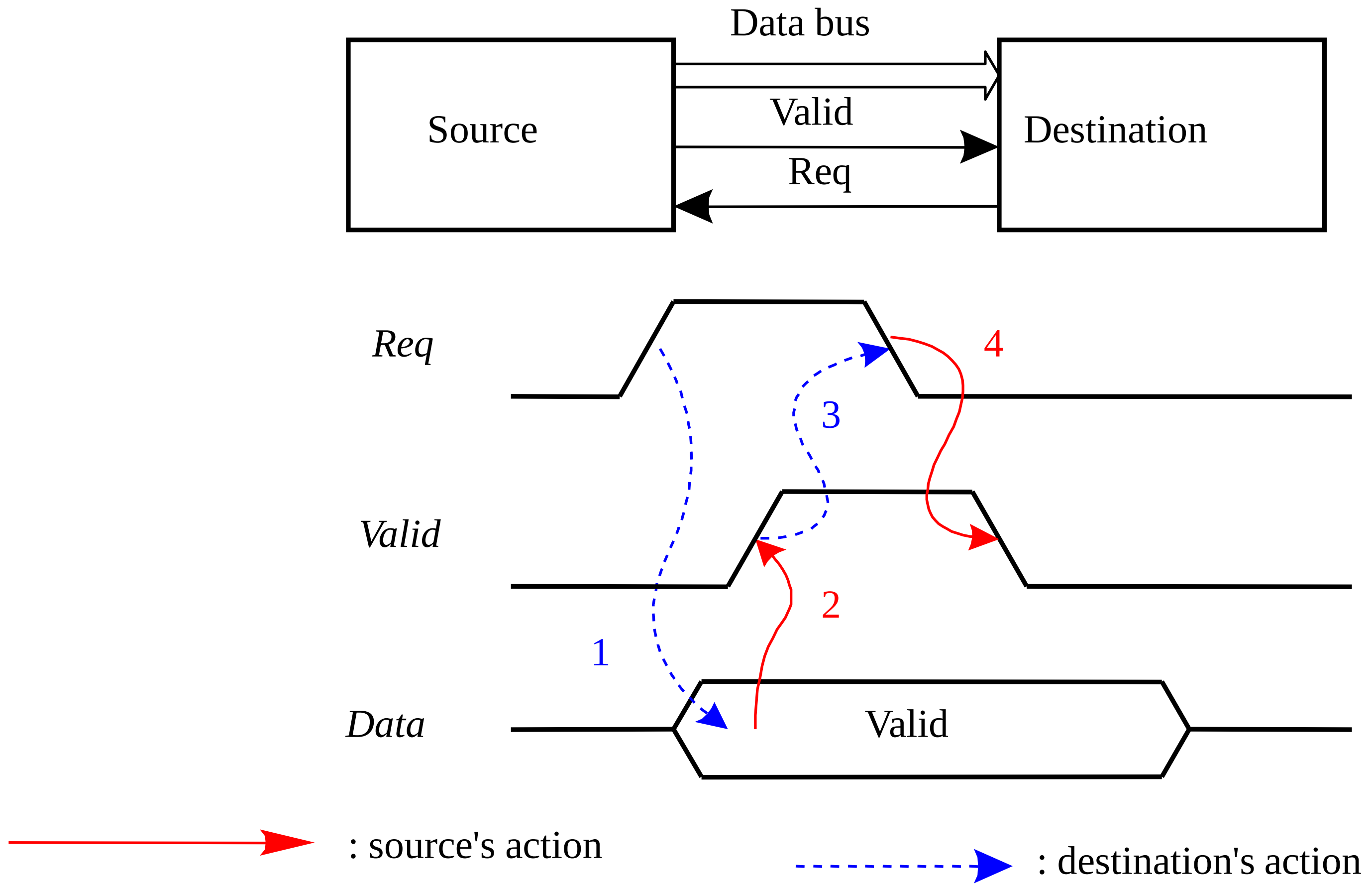
1. ready:
2. data valid:
3. data acceptance:
4. acknowledge:



Source-initiated transfer

- In the handshaking transfer, four events are proceeded in a cycle order:
  1. ready: The destination device deasserts the acknowledge signal and is ready to accept the next data.
  2. data valid: The source device places the data onto the data bus and asserts the valid signal to notify the destination device that the data on the data bus is valid.
  3. data acceptance: The destination device accepts (latches) the data from the data bus and asserts the acknowledge signal.
  4. acknowledge: The source device invalidates data on the data bus and deasserts the valid signal

Destination-initiated transfer

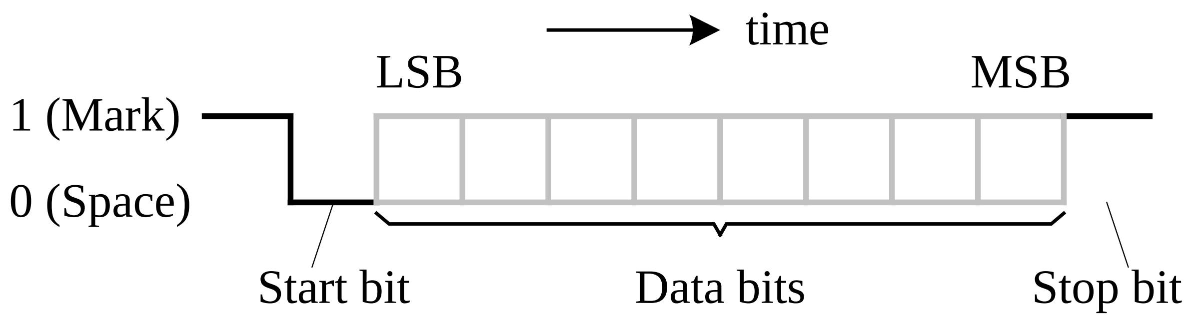


Destination-initiated transfer

- In the handshaking transfer, four events are proceeded in a cycle order:
  1. request: The destination device asserts the request signal to request data from the source device.
  2. data valid: The source device places the data on the data bus and asserts the valid signal to notify the destination device that the data is valid now.
  3. data acceptance: The destination device accepts (latches) the data from the data bus and asserts the request signal.
  4. acknowledge: The source device invalidates data on the data bus and deasserts the valid signal to notify the destination device that it has removed the data from the data bus

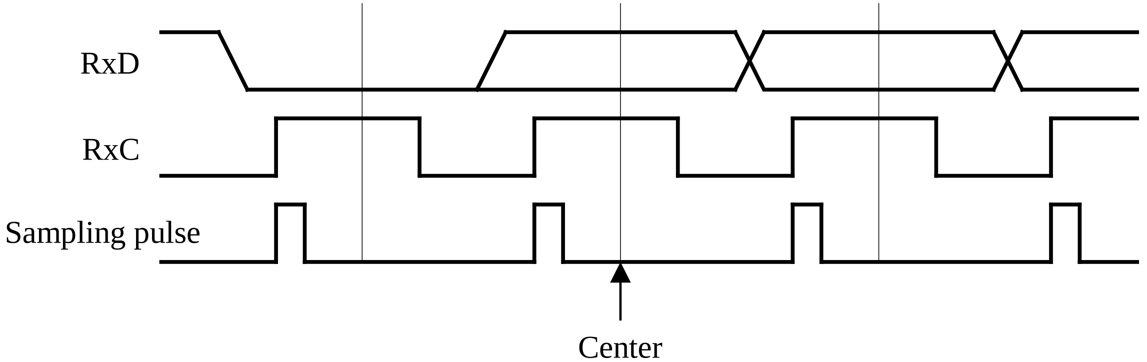
Asynchronously Serial Data Transfers

- The clock signal is not sent with data.
- The receiver generates its local clock that is then used to capture the data being received.
- When there is no data to be sent, the transmitter continuously sends 1s in order to maintain a continuous communication channel.
- The receiver monitors the channel continuously until the start bit is detected.

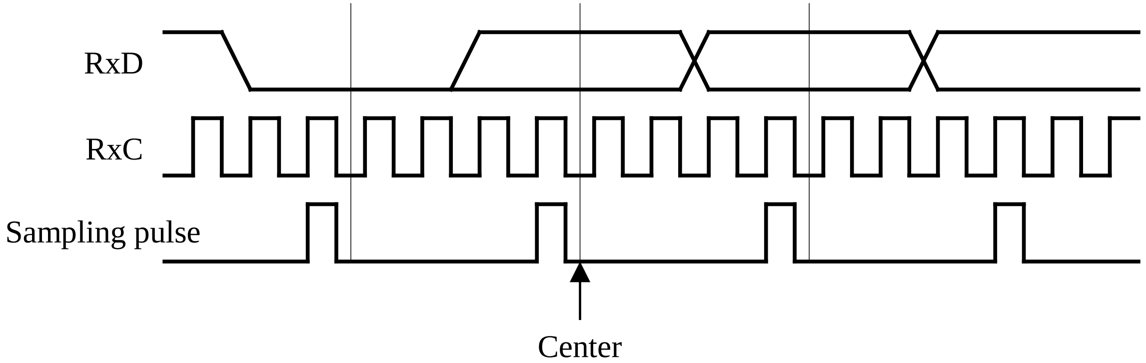


Asynchronously Serial Data Transfers

- Sampling at the same frequency



- Sampling using 4 times frequency

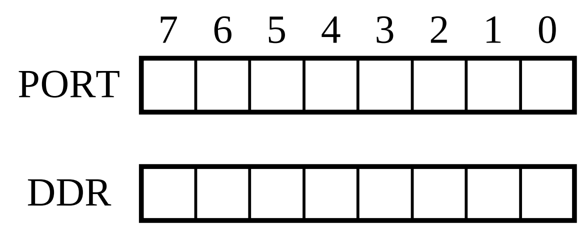


General-Purpose Input and Output Devices

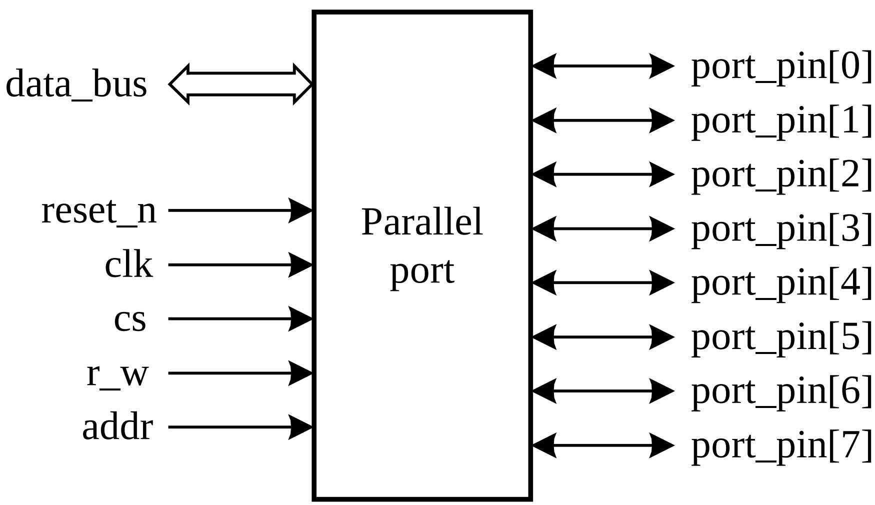
- The general-purpose input and output (GPIO) is a device that can be programmed into
  - input
  - output
  - bidirectional

General-Purpose Input and Output Devices

- An example of 8-bit GPIO



(b) Programming model



(a) Hardware model

cs	r_w	addr	Function
0	ϕ	ϕ	Chip unselected
1	1	0	Read PORT
1	0	0	Load PORT
1	1	1	Read DDR
1	0	1	Load DDR

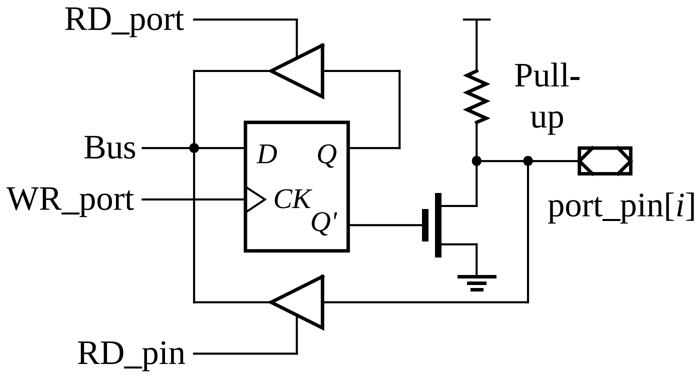
(c) Function table

General-Purpose Input and Output Devices

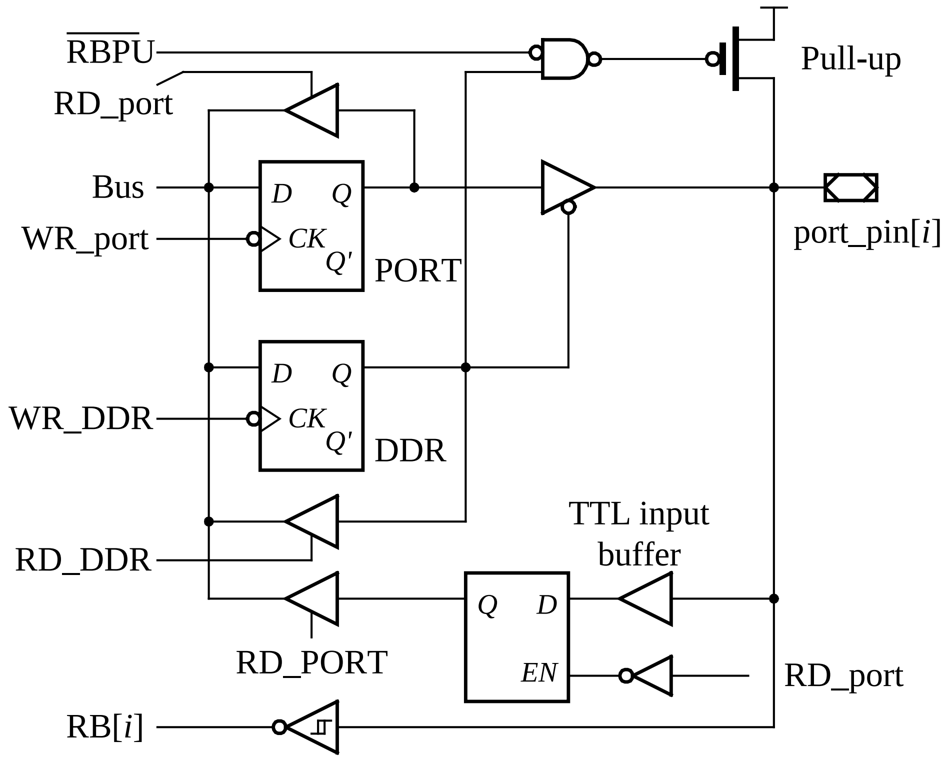
- Design issues:
  - Readback capability of PORT register
  - Group or individual bit control
  - Selection the value of DDR
  - Handshaking control
  - Readback capability of DDR
  - Input latch
  - Input/Output pull-up
  - Drive capability

General-Purpose Input and Output Devices

❖ The logic diagram of the *i*th-bit of two GPIO examples.



(a) MCS-51 Port 1



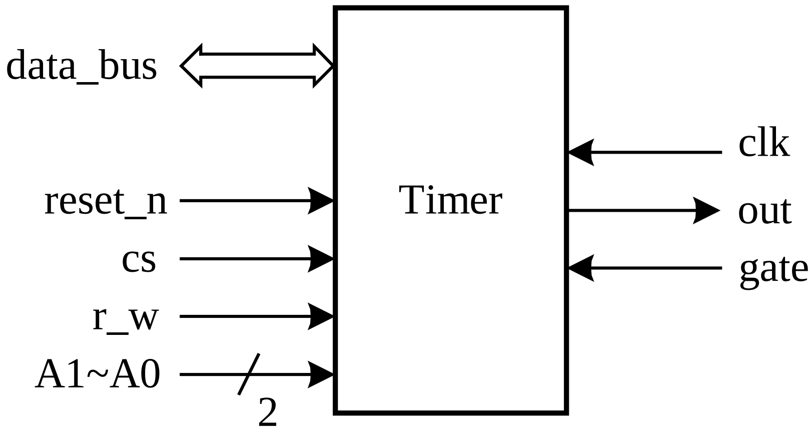
(b) PIC 18Fxx2 Port RB2:RB0

Timers

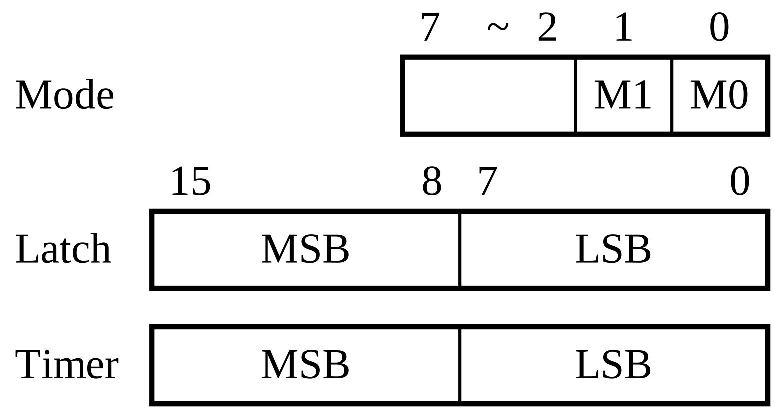
- Important applications:
  - time-delay creation
  - event counting
  - time measurement
  - period measurement
  - pulse-width measurement
  - time-of-day tracking
  - waveform generation
  - periodic interrupt generation

Timers

The latch register stores the initial value to be loaded into the timer for counting down.



(a) Hardware model



(b) Programming model

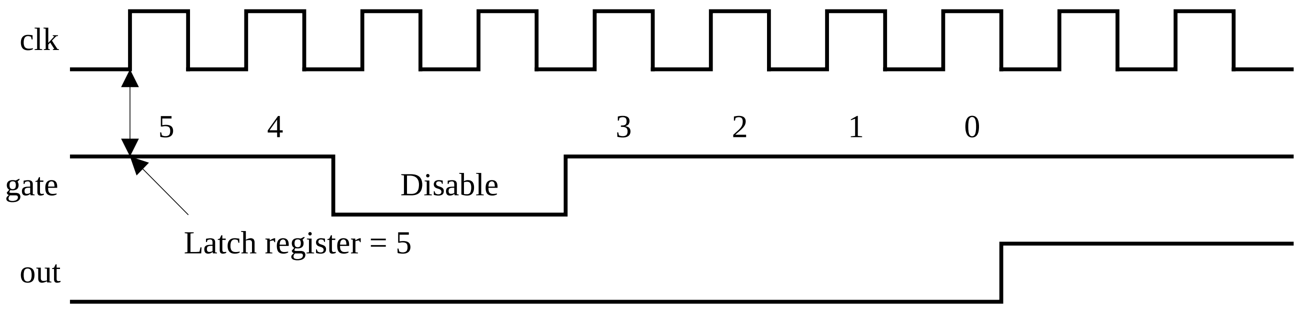
cs	r_w	A1	A0	Function
0	ϕ	ϕ	ϕ	Chip unselected
1	1	0	0	Read Latch register
1	1	0	1	Read timer register
1	0	0	0	Write Latch register
1	0	1	0	Write mode register

(c) Function table

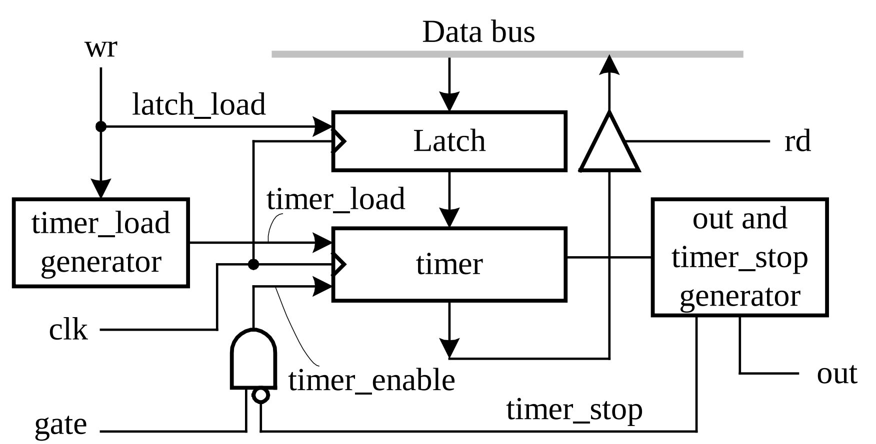
Basic Timer Operations

- A counter is called a timer if it is operated at a known clock of fixed frequency.
- In practice, the timers used in most µC systems are counters with programmable operation modes.
- The basic operation modes of a timer are as follows:
  - terminal count (binary/BCD event counter)
  - rate generation
  - (digital) monostable (or called one-shot)
  - square-wave generation

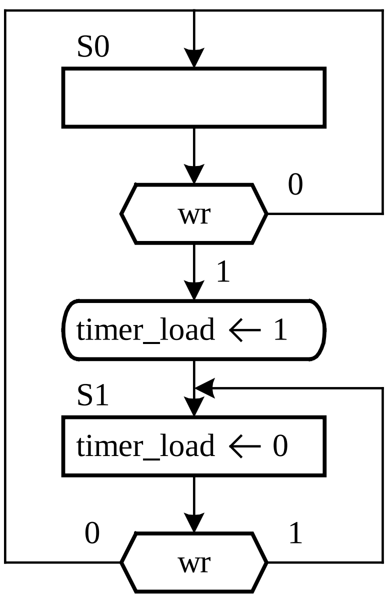
Terminal Count



(a) A waveform example of terminal-count mode



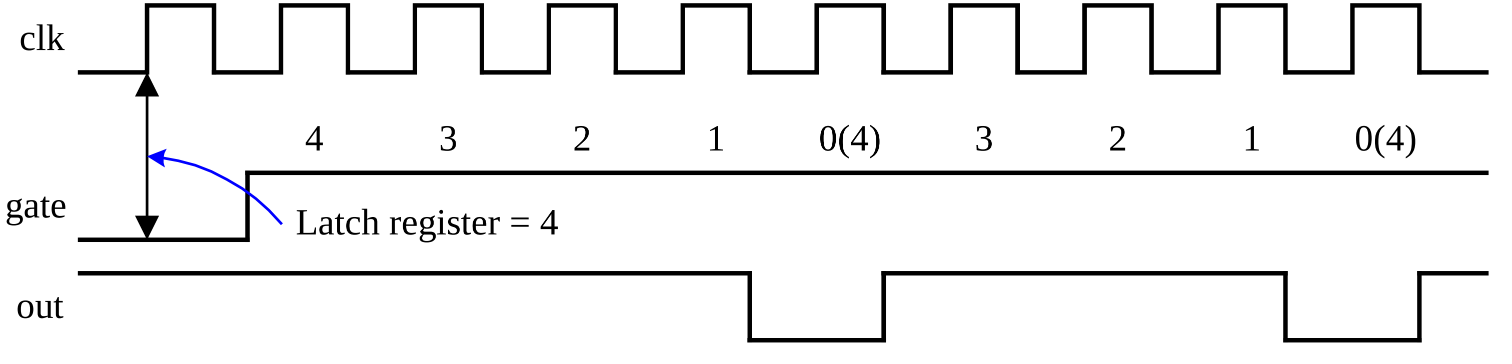
(b) Block diagram of terminal-count mode



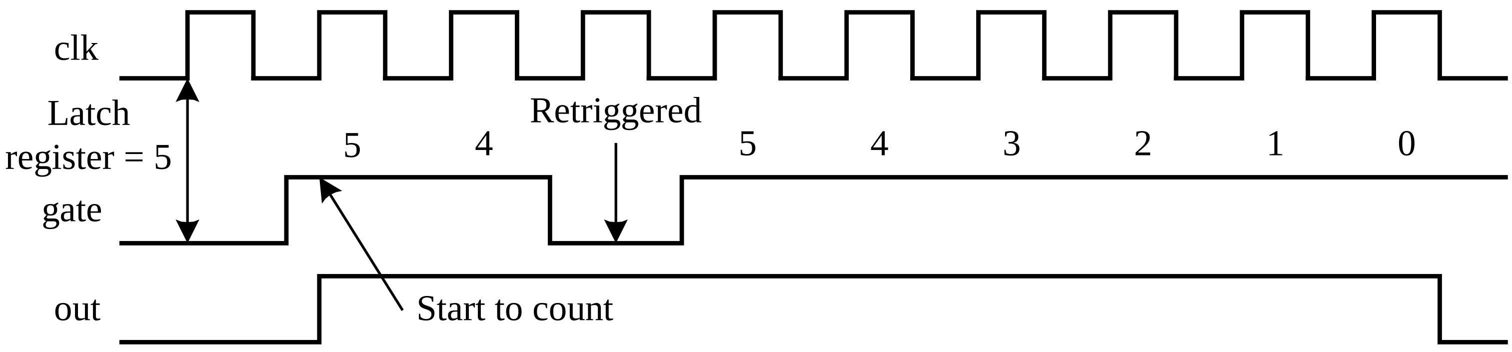
(c) Generate one-cycle timer_load pulse

Rate Generation

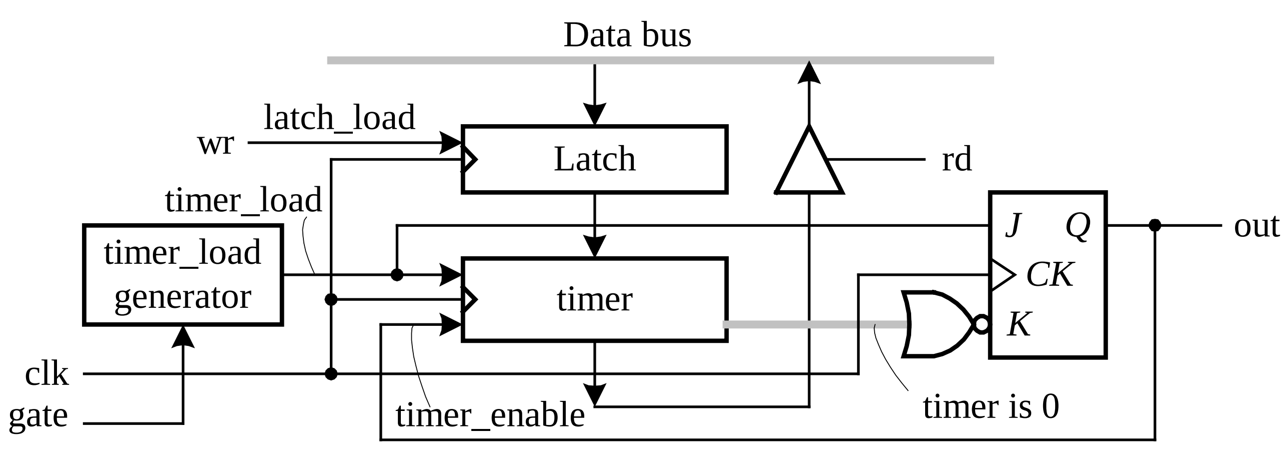
- The out terminal outputs one clock pulse for every *N* clock pulses.
- The gate input should be fixed at the logic 1 to enable the timer.
- It is implemented by being reloaded the timer register from latch register whenever the terminal count is reached.



Retriggerable Monostable (One-Shot) Operation



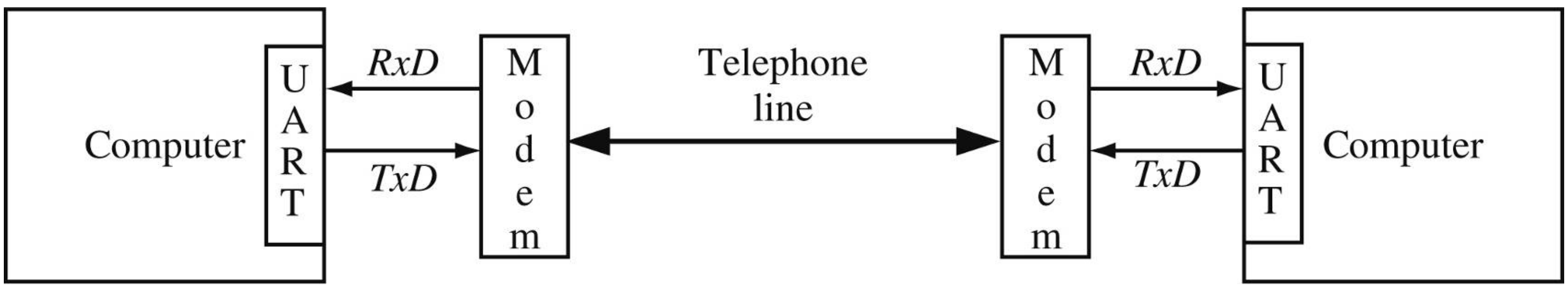
(a) A waveform example of one-shot mode

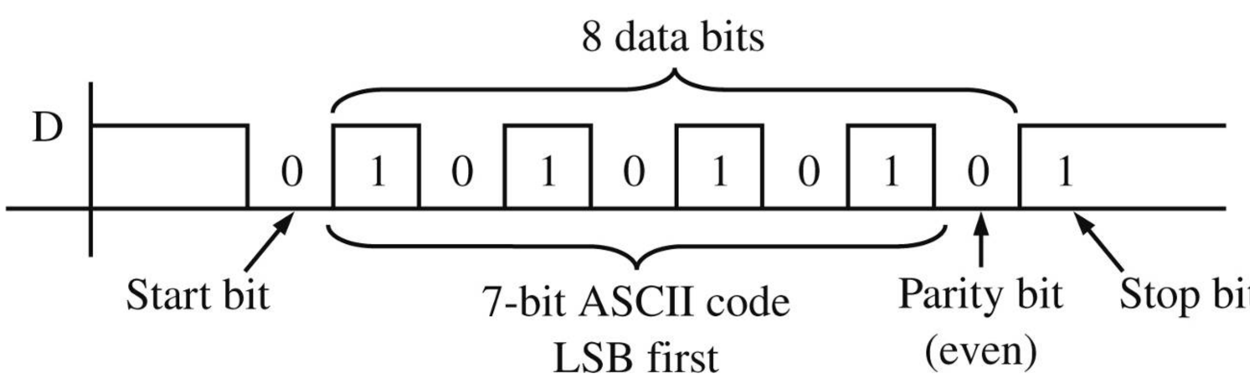


(b) Block diagram of one-shot mode

UART

- Universal Asynchronous Receiver Transmitter
  - Serial Data Transmission

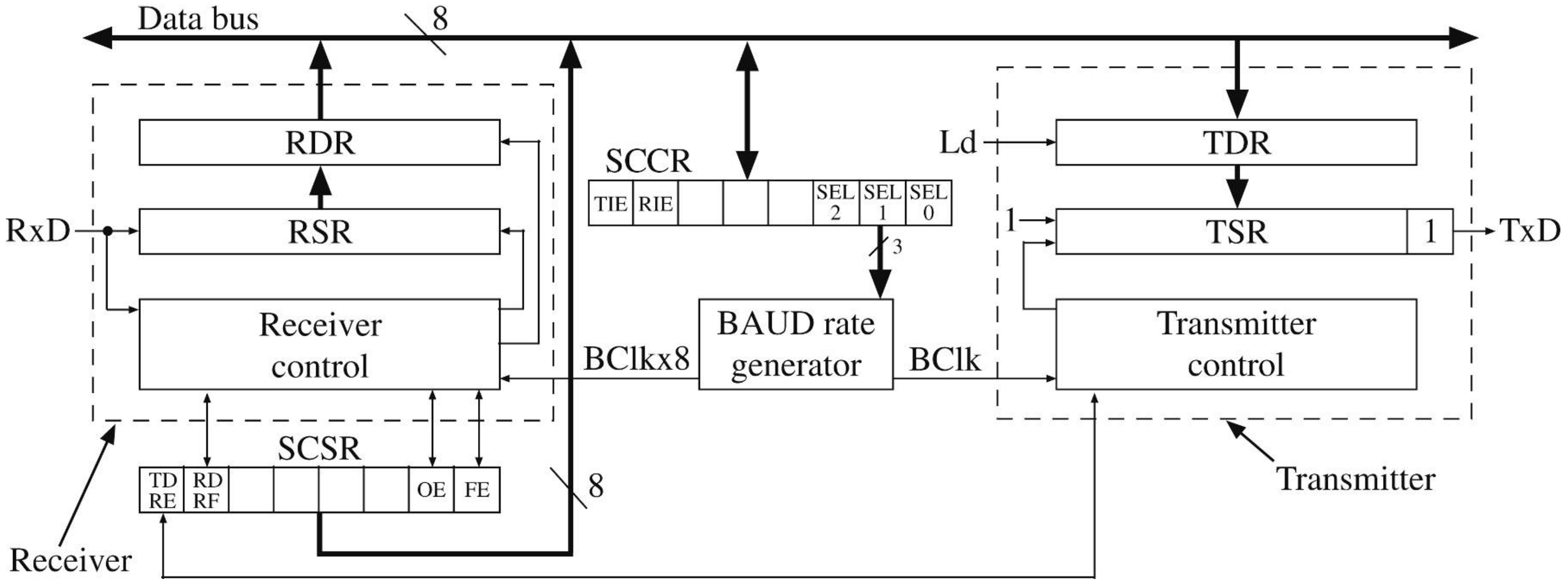




68HC11 Microcontroller

- UART Registers
  - RSR Receive Shift Register
  - RDR Receive Data Register
  - TDR Transmit Data Register
  - TSR Transmit Shift Register
  - SCCR Serial Communications Control Register
  - SCSR Serial Communications Status Register
- UART Flags
  - TDRE Transmit Data Register Empty
  - RDRF Receive Data Register Full

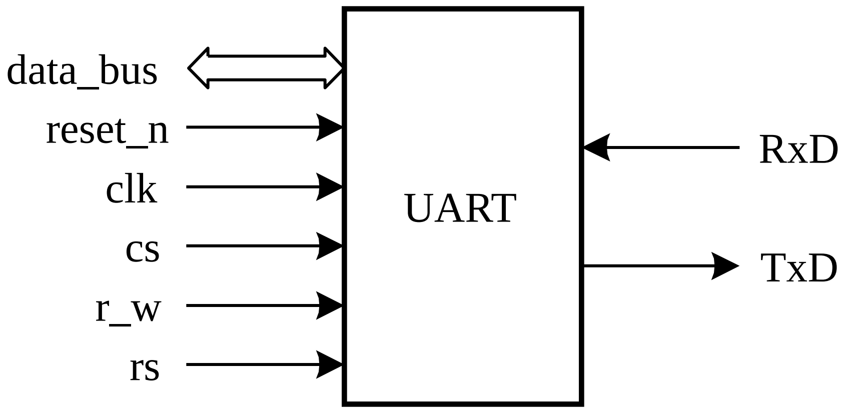
UART Block Diagram



UARTs

- UART is a device used to provide serial data ports used to communicate with serial devices.
- The hardware model includes
  - the CPU interface
  - the I/O interface
- The software model consists of four registers:
  - receiver data register (RDR)
  - transmitter data register (TDR)
  - status register (SR)
  - control register (CR)

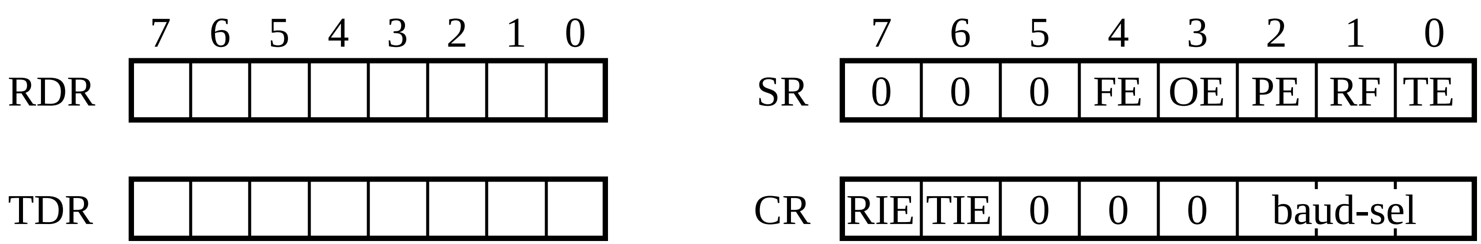
UARTs



(a) Hardware model

cs	r_w	rs	Function
0	ϕ	ϕ	Chip unselected
1	1	0	Read SR
1	0	0	Write CR
1	1	1	Read RDR
1	0	1	Write TDR

(b) Function table



(c) Programming model

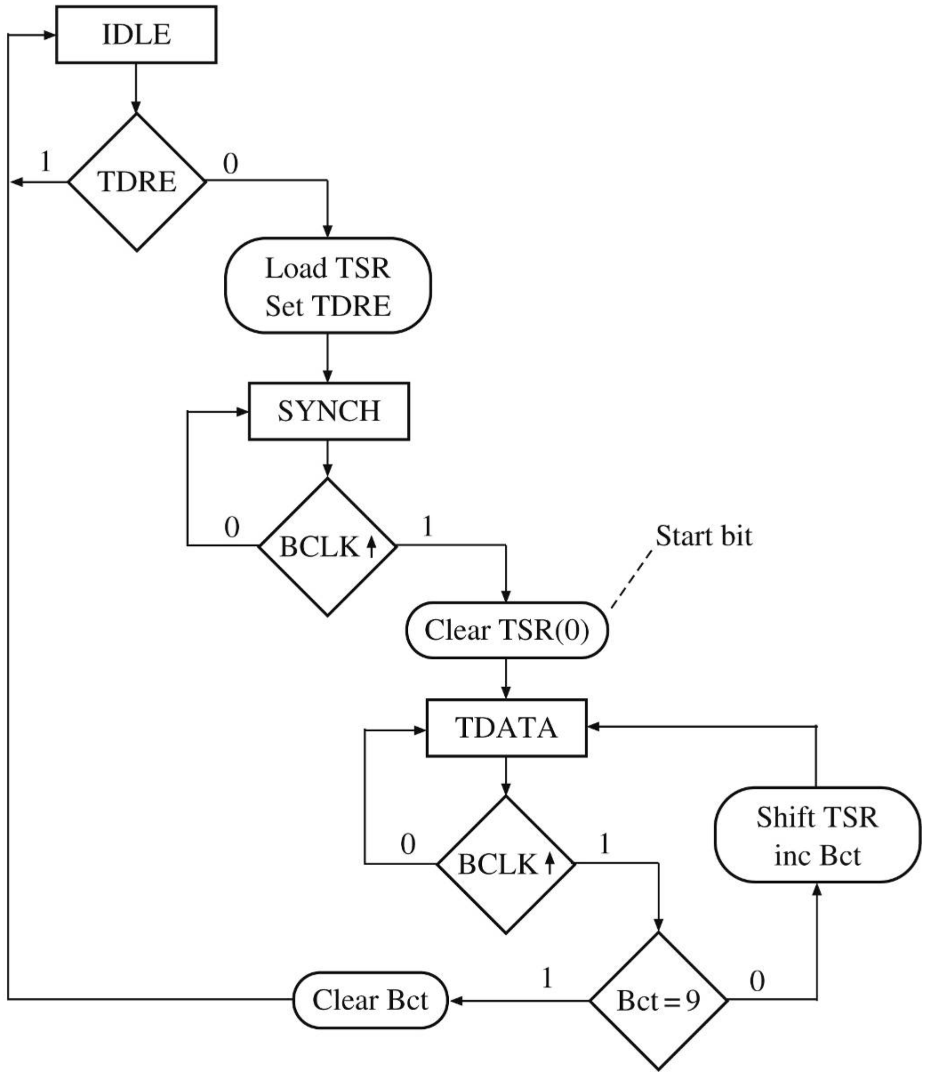
UARTs

- Design issues
  - baud rate: 300 to 19200, or even more
  - sampling clock frequency: RxC = n TxC (n= 1,4,16,64)
  - stop bits: 1, 1.5, 2 bits
  - parity check
    - Even: the number of 1s of information and parity is even.

Transmitter Operation

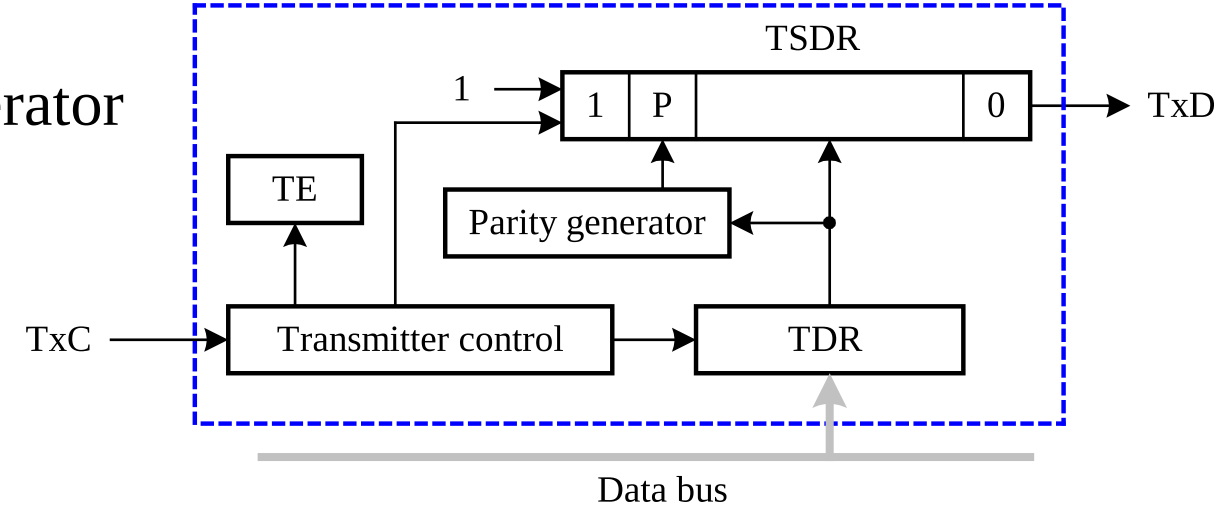
- Microcontroller waits until TDRE = '1'
  - Loads data into TDR
  - Clears TDRE
- UART transfers data from TDR to TSR
  - Sets TDRE
- UART outputs start bit ('0') then shifts TSR right eight times followed by a stop bit ('1')

Transmitter SM Chart

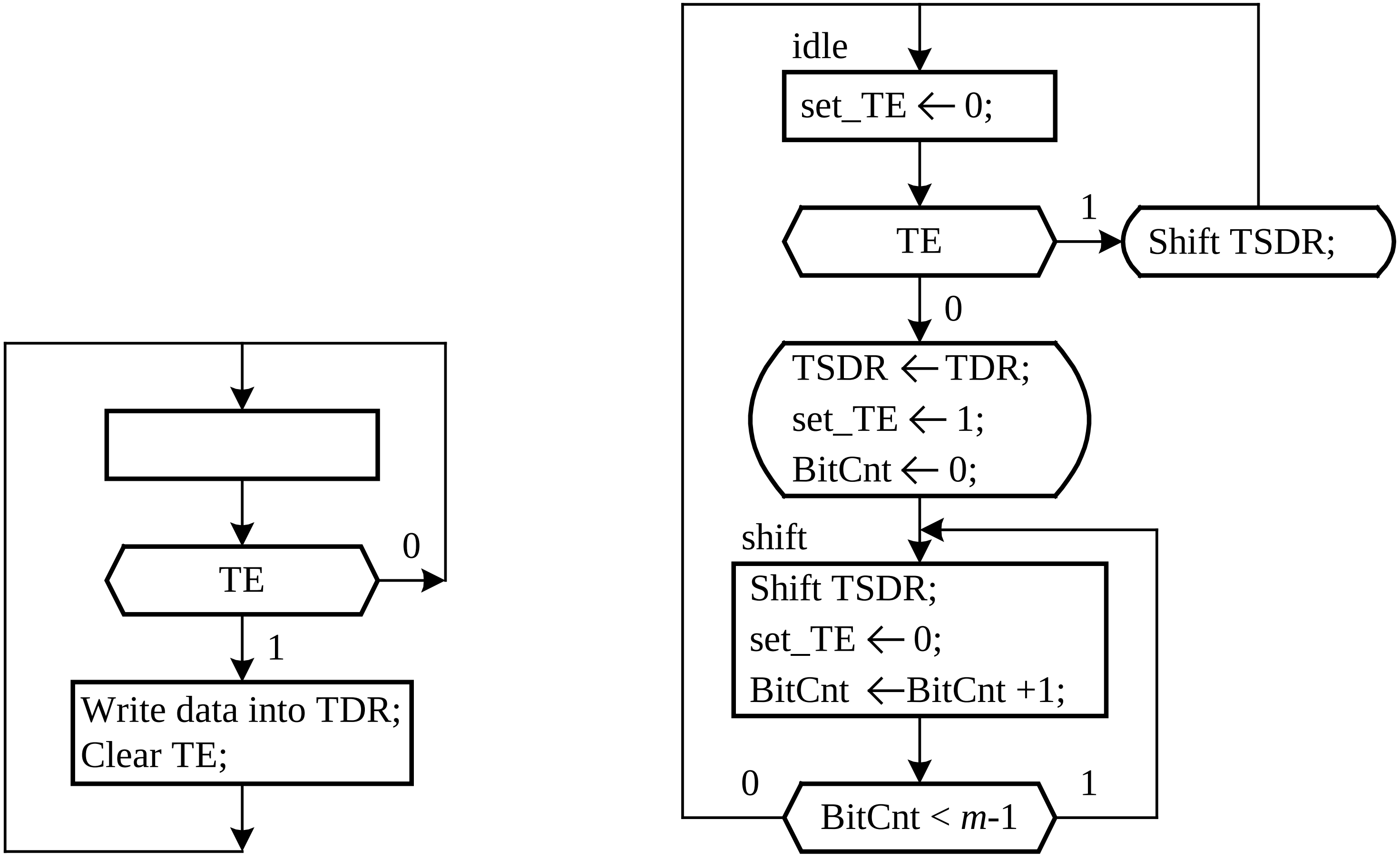


A Transmitter of UARTs

- The essential component of the transmitter is a shift register.
- The transmitter is composed of
  - a transmitter shift data register (TSDR)
  - a TDR empty flag (TE)
  - a transmitter control circuit
  - a TDR
  - parity generator



A Transmitter of UARTs



(a) CPU operations (b) TSDR operations

UART transmitter

```
// an example of UART transmitter
module UART_transmitter(clk, reset_n, load_TDR, data_bus,
                        TE, TxC, TxD);
parameter M = 10; //the frame size excluding the stop bit
input clk, reset_n, TxC, load_TDR;
input [7:0] data_bus;
output TxD, TE;
// internal used registers
reg [7:0] TDR;
reg [M-1:0] TSDR;
reg [3:0] BitCnt;
reg ps, ns, set_TE, TE;
localparam idle = 1'b0, shift = 1'b1;
// load TDR when load_TDR is activated
always @(posedge clk  or negedge reset_n)
   if (!reset_n) TDR <= 8'b0;
   else if (load_TDR) TDR <= data_bus;
// update TDR empty flag (TE)
always @(posedge clk or negedge reset_n)
   if (!reset_n) TE <= 1'b1;
   else TE <= (set_TE && !TE) || (!load_TDR && TE);
// load TSDR from TDR and perform data transmission
// step 1: initialize and update state registers
always @(posedge TxC or negedge reset_n)
```

UART transmitter

```
      if (!reset_n) ps <= idle;
      else ps <= ns;
   // step 2: compute next state
   always @(*)
      case (ps)
         idle: if (TE == 1) ns = idle;
               else ns = shift;
         shift: if (BitCnt < M - 1) ns = shift;
               else ns = idle;
      endcase
   // step 3: execute RTL operations
   always @(posedge TxC or negedge reset_n)
      if (!reset_n) begin TSDR <= {M{1'b1}};
         BitCnt <= 0; set_TE <= 1'b0; end
      else case (ps)
         idle: begin set_TE <= 1'b0;
               if (TE == 1) TSDR <= {1'b1, TSDR[M-1:1]};
               else begin
                  TSDR <= {^TDR, TDR, 1'b0};
                  set_TE <= 1'b1;
                  BitCnt <= 0; end  end
         shift: begin
                  TSDR <= {1'b1, TSDR[M-1:1]};
                  set_TE <= 1'b0;
                  BitCnt <= BitCnt + 1; end
      endcase
   assign TxD = TSDR[0] & (ps == shift) | (ps == idle);
   endmodule
```

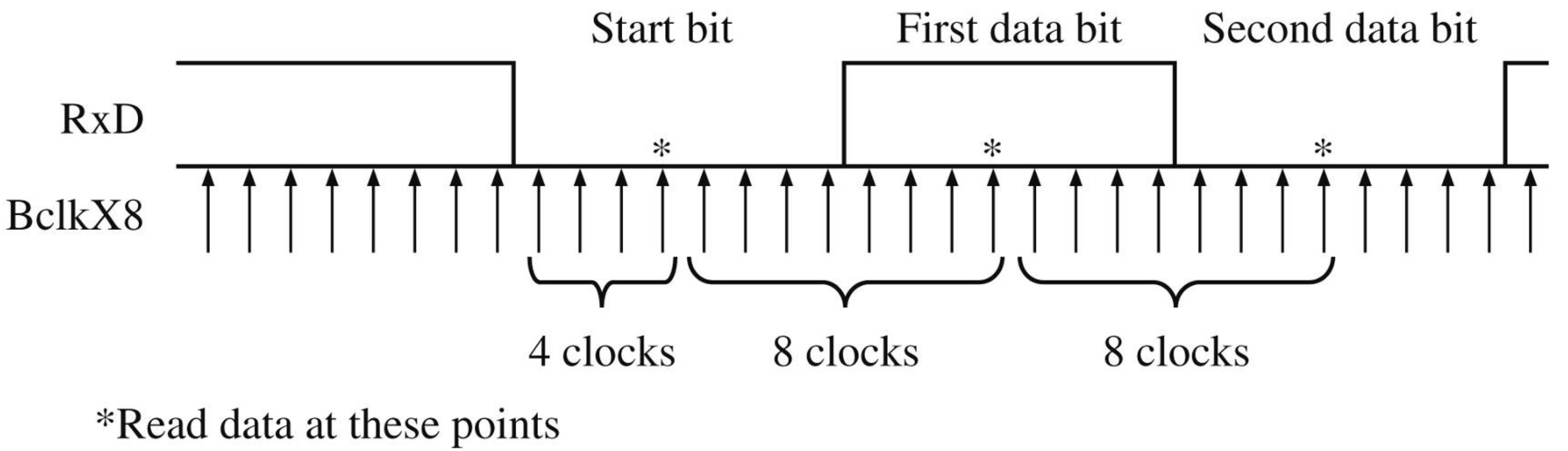
even parity (^TDR)

start bit (1'b0)

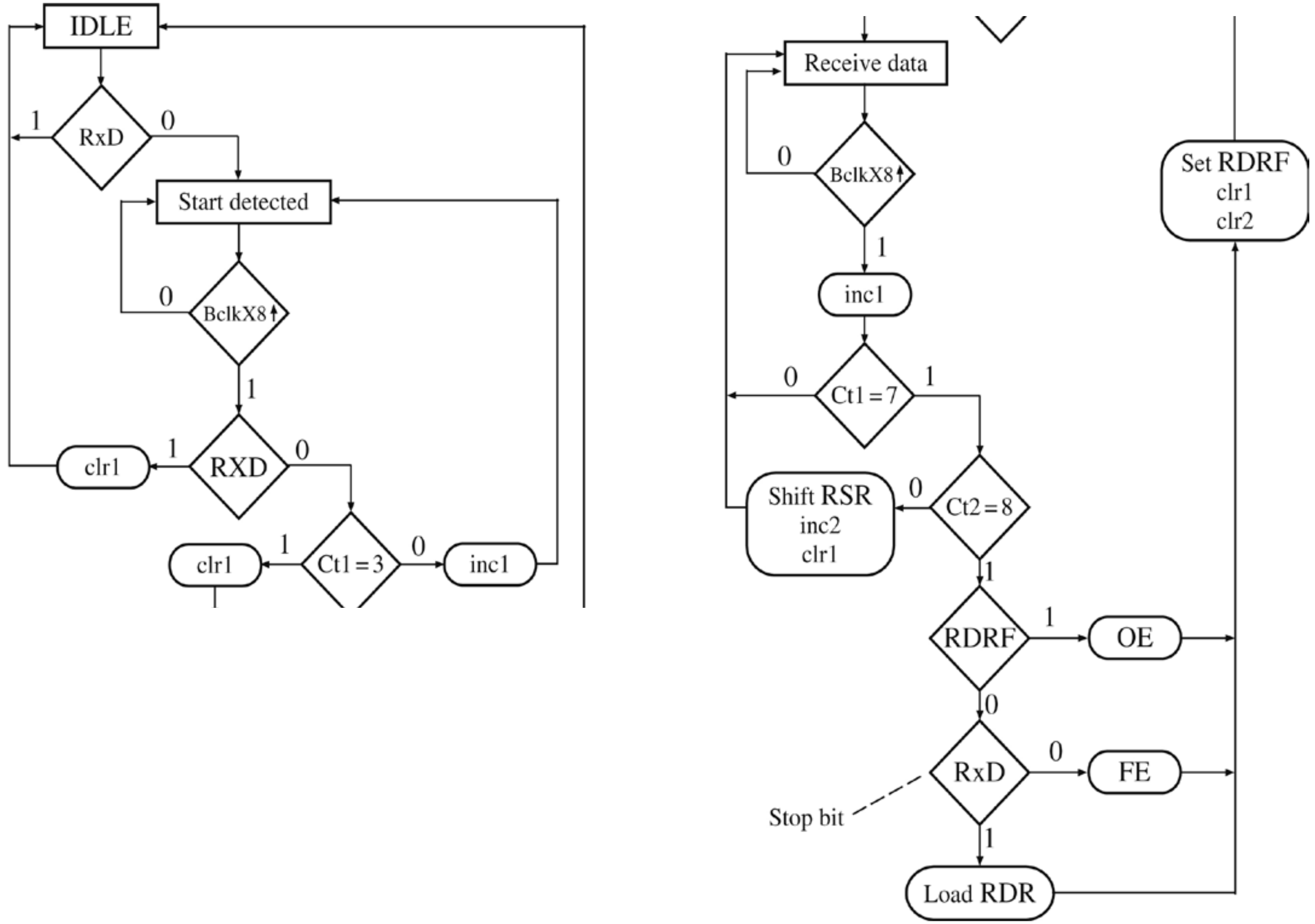
Receiver Operation

- UART waits for start bit
  - Shifts bits into RSR
- When all data bits and stop bit are received
  - RSR loaded into RDR
  - Set RDRF
- Microcontroller waits until RDRF is set
  - Read RDR
  - Clear RDRF

Sampling RxD

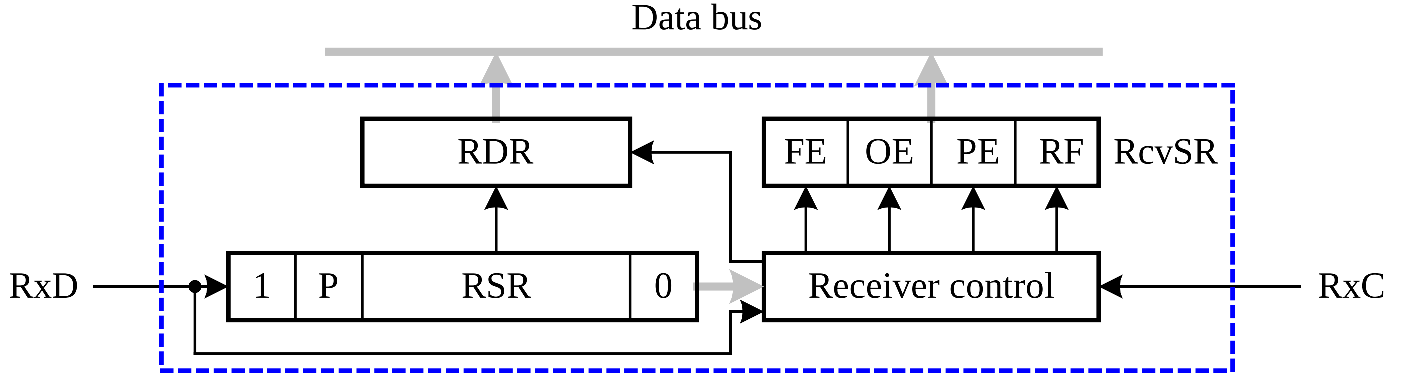


Receiver SM Chart

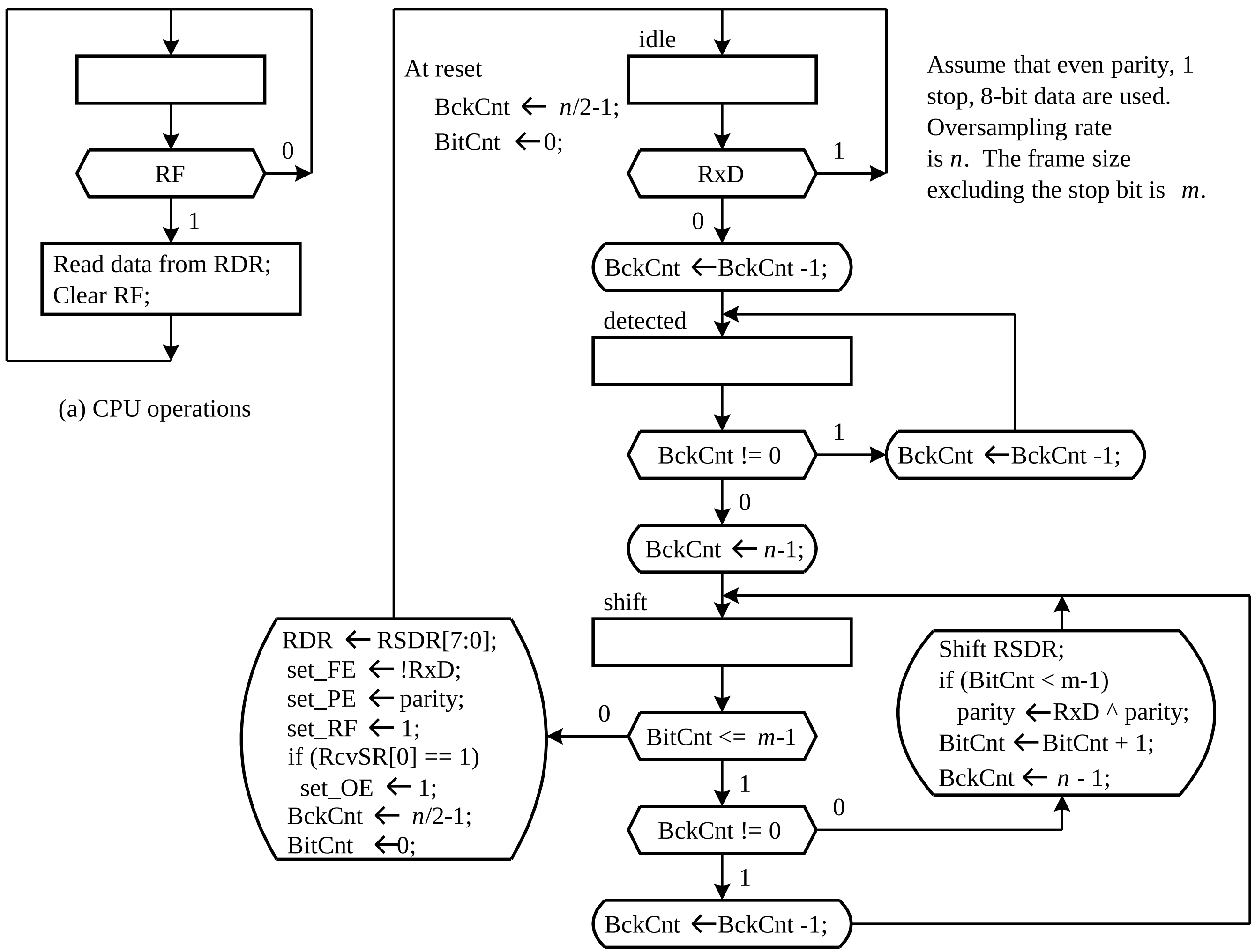


A Receiver of UARTs

- The essential component of the receiver is also a shift register.
- The receiver is composed of
  - a RDR
  - a receiver shift data register (RSDR)
  - a status register
  - a receiver control circuit



A Receiver of UARTs



(b) RSDR operations

UART receiver

```
// an example of UART receiver -- RxC is 8 times TxC
// 1 stop, 8-bit data, and even parity are used
module UART_receiver(clk, reset_n, read_RDR, RDR, RcvSR,
                     RxC, RxD);
parameter M = 10; // default frame size excluding the
                  // stop bit
parameter N = 8;  // default sampling clock frequency
localparam K = 3; // K = log2 N
localparam L = 4; // L = log2 M
input  clk, reset_n, read_RDR, RxC, RxD;
output reg [3:0] RcvSR;
output reg [7:0] RDR; // receiver data register
// internal used registers
reg [M-1:0] RSDR;   // receiver shift data register
reg [L-1:0] BitCnt; // number of bits received
reg [K-1:0] BckCnt; // number of TxC clocks elapsed
reg [1:0] ps, ns;
wire RcvSR_reset;
reg parity, set_FE, set_OE, set_PE, set_RF, set_RF_1clk;
```

UART receiver

```
localparam idle = 2'b00, detected = 2'b01, shift = 2'b10;
// update status register (RcvSR)
assign RcvSR_reset = !reset_n || read_RDR;
always @(posedge clk  or posedge RcvSR_reset)
   if (RcvSR_reset) RcvSR <= 4'b0000;
   else if (set_RF_1clk) RcvSR <= {set_FE, set_OE, set_PE,
                                    set_RF};
// generate set_RF_1clk signal from set_RF
reg state;
localparam S0 = 1'b0, S1 = 1'b1;
always @(posedge clk or negedge reset_n)
   if (!reset_n) begin set_RF_1clk <= 1'b0;
                       state <= S0; end
   else case (state)
      S0: if (set_RF) begin state <= S1;
                 set_RF_1clk <= 1'b1; end
          else state <= S0;
      S1: begin set_RF_1clk <= 1'b0;
          if (set_RF) state <= S1; else state <= S0; end
   endcase
// receive data and load RSDR into RDR
// step 1: initialize and update state registers
always @(posedge RxC or negedge reset_n)
   if (!reset_n) ps <= idle;
   else ps <= ns;
// step 2: compute next state
```

UART receiver

```
always @(*)
   case (ps)
      idle: if (RxD == 1) ns = idle;
            else ns = detected;
      detected: if (BckCnt != 0) ns = detected;
            else ns = shift;
      shift: if (BitCnt <= M-1)  ns = shift;
            else ns = idle;
      default: ns = idle;
   endcase
// step 3: execute RTL operations
always @(posedge RxC or negedge reset_n)
   if (!reset_n) begin RSDR <= {M{1'b0}};
     RDR <= 8'b00000000;
     BckCnt <= N/2-1; BitCnt <= 0; set_FE <= 1'b0;
     set_OE <= 1'b0; set_PE <= 1'b0;
   set_RF <= 1'b0; parity <= 0; end
   else case (ps)
```

UART receiver

```
idle: begin set_FE <= 1'b0; set_OE <= 1'b0;
            set_PE <= 1'b0;
            set_RF <= 1'b0; parity <= 1'b0;
            if (RxD == 0) BckCnt <= BckCnt - 1; end
detected: if (BckCnt != 0) BckCnt <= BckCnt - 1;
            else BckCnt <= N - 1;
shift: begin
            if (BitCnt <= M - 1) begin
               if (BckCnt != 0)
                  BckCnt <= BckCnt - 1;
               else begin
                  RSDR <= {RxD, RSDR[M-1:1]};
                  if (BitCnt < M-1)
                     parity <= RxD ^ parity;
                  BitCnt <= BitCnt + 1;
                  BckCnt <= N - 1;  end end
            else begin RDR <= RSDR[7:0];
               set_FE <= !RxD; set_PE <= parity;
               BckCnt <= N/2 - 1; set_RF <= 1'b1;
               if (RcvSR[0] == 1) set_OE <= 1'b1;
               BitCnt <= 0; end
         end
   endcase
endmodule
```

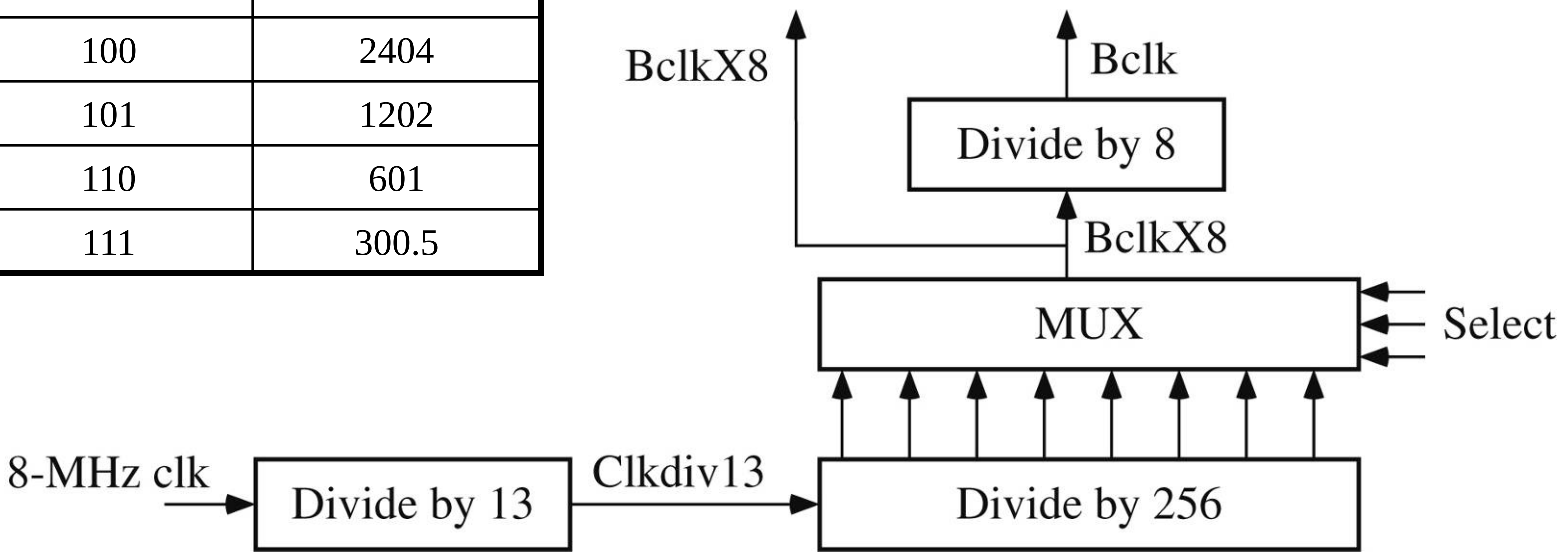
shift

Baud-Rate Generators

- The baud-rate generator provides both clock sources: TxC and RxC, for transmitter and receiver modules, respectively.
- Two most widely used approaches to designing baud-rate generators:
  - Multiplexer-based approach
  - timer-based approach

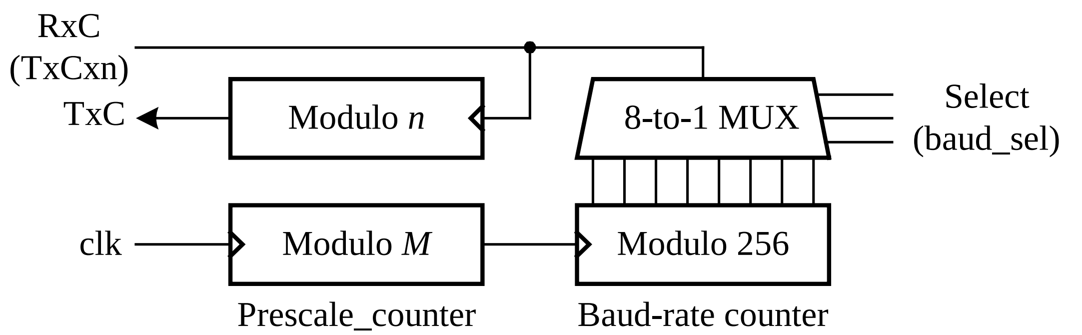
Baud Rate Generator

Select Bits	BAUD Rate
000	38,462
001	19,231
010	9615
011	4808
100	2404
101	1202
110	601
111	300.5

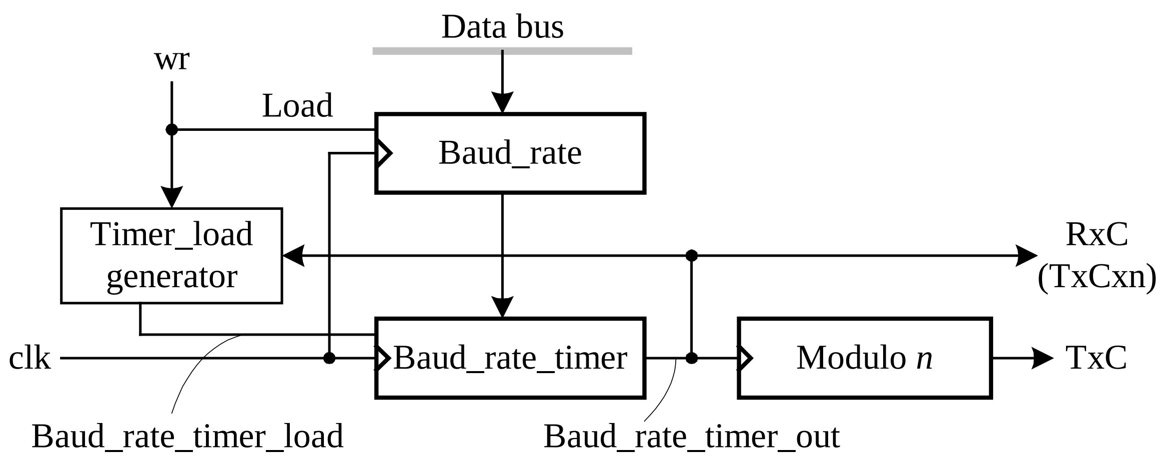


300.5 x 8 x 256 x 13 =8000512

Baud-Rate Generators



(a) Multiplexer-based baud-rate generator



(b) Timer-based baud-rate generator

Baud-rate generator

```
// an example of multiplexer-based baud-rate generator
module UART_baud_rate(clk, reset_n, baud_sel, TxC, TxCx8);
parameter BD_STEPS = 8; // default number of baud-rate
                        // steps
input clk, reset_n;
input [2:0] baud_sel;
output TxC, TxCx8;
reg TxCx8;
integer i;
// internal used reg variables
wire prescale_out, TxC, TxCx8_clk;
reg  [2:0] prescale_counter;
reg  [BD_STEPS-1:0] baud_rate_counter;
```

Baud-rate generator

```
reg  [2:0] mod8_counter;
// describe the prescale_counter, a presettable down
// counter
always @(posedge clk or negedge reset_n)
   if (!reset_n) prescale_counter <= 0;
   else prescale_counter <= prescale_counter + 1;
assign prescale_out = &prescale_counter;
// describe the a modulo-256 up counter
always @(posedge clk or negedge reset_n)
   if (!reset_n) baud_rate_counter <= 0;
   else if (prescale_out)
            baud_rate_counter <= baud_rate_counter + 1;
// describe the 8-to-1 multiplexers for selecting the
// required baud rate
always @(baud_sel or baud_rate_counter) begin
   TxCx8 = 0;
   for (i = 0; i < BD_STEPS; i = i + 1)
      if (baud_sel == i) TxCx8 = baud_rate_counter[i];
   end
assign TxCx8_clk = TxCx8;
// describe the modulo8_counter
always @(posedge TxCx8_clk or negedge reset_n)
   if (!reset_n) mod8_counter <= 0;
   else mod8_counter <= mod8_counter + 1;
assign TxC = mod8_counter[2];
endmodule
```

UART top

```
// instantiate transmitter, receiver, and baud-rate
// generator modules
UART_transmitter my_transmitter (.clk(clk), .reset_n
  (reset_n), .load_TDR(load_TDR), .data_bus(data_bus),
  .TE(TE), .TxC(TxC), .TxD(TxD));
UART_receiver my_receiver (.clk(clk), .reset_n(reset_n),
  .read_RDR(read_RDR), .RDR(RDR), .RcvSR (RcvSR),
  .RxC(RxC), .RxD(RxD));
UART_baud_rate my_baud_rate(.clk(clk), .reset_n(reset_n),
        .baud_sel(CR[2:0]), .TxC(TxC), .TxCx8(RxC));

// generate interface control signals
assign read_UART  = (cs == 1) && (r_w == 1),
       write_UART = (cs == 1) && (r_w == 0);
assign read_RDR = read_UART && (rs == 1'b1),
       load_TDR = write_UART && (rs == 1'b1),
       read_SR  = read_UART && (rs == 1'b0),
       load_CR  = write_UART && (rs == 1'b0);
```

cs	r_w	rs	Function
0	ϕ	ϕ	Chip unselected
1	1	0	Read SR
1	0	0	Write CR
1	1	1	Read RDR
1	0	1	Write TDR

(b) Function table

UART top

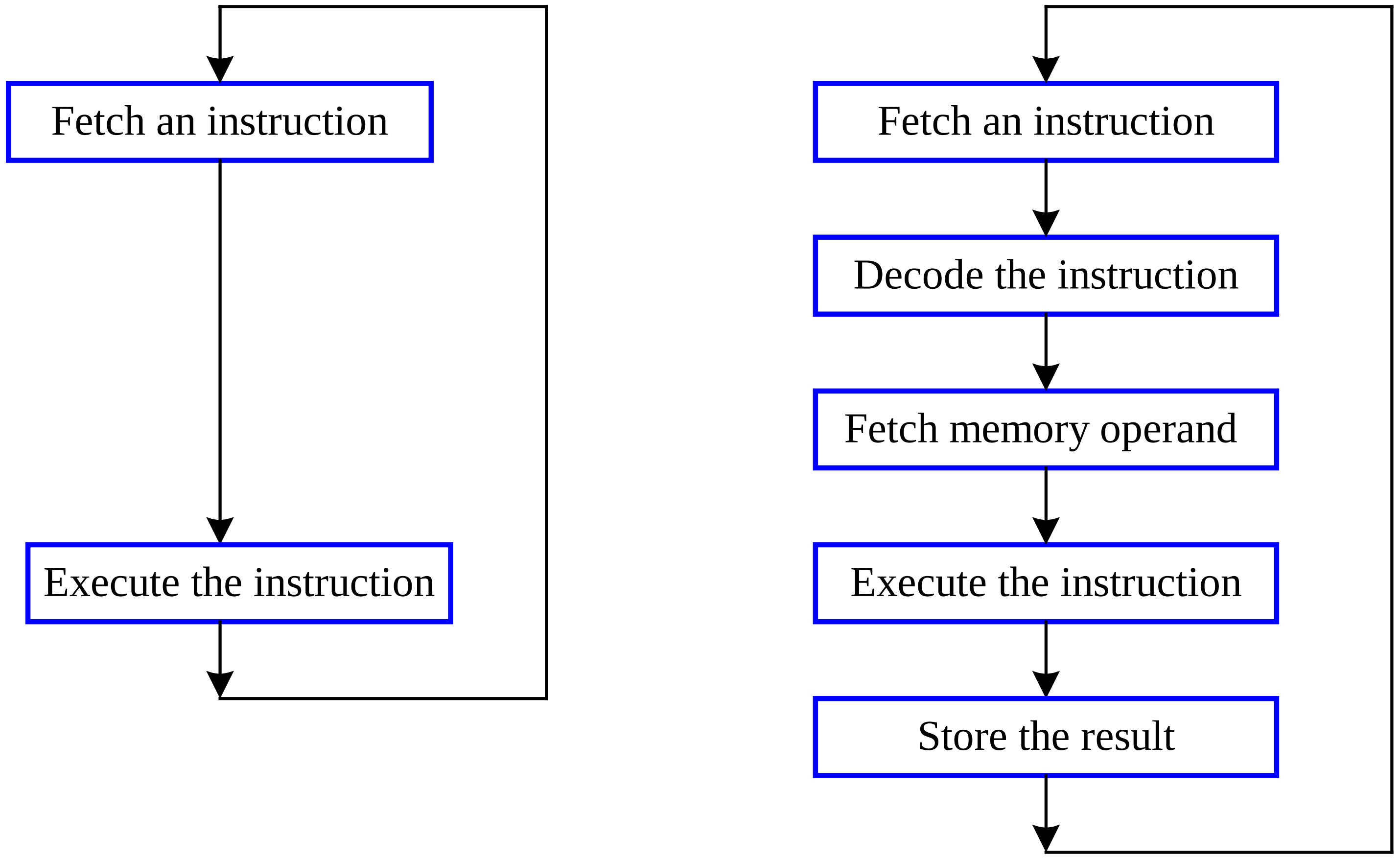
```
// read RDR or SR
assign data_bus = (read_RDR) ? RDR :
                  ((read_SR) ? {3'b000, RcvSR, TE}: 8'bz);
// define the control register
always @(posedge clk or negedge reset_n)
   if (!reset_n) CR <= 8'b0000_0000; // control register
   else if (load_CR) CR <= data_bus;

// irq generation logic
always @(posedge clk or negedge reset_n)
   if (!reset_n) IRQ <= 1'b0;
   else if (((CR[7] == 1) && (RcvSR[0] == 1 ||
             RcvSR[2] == 1)) ||
            (CR[6] == 1) && (TE == 1)) IRQ <= 1'b1;
        else IRQ <= 1'b0;
endmodule
```

RIE (CR[7])

RF (RcvSR[0])

TIE (CR[6])

CPU Basic Operations

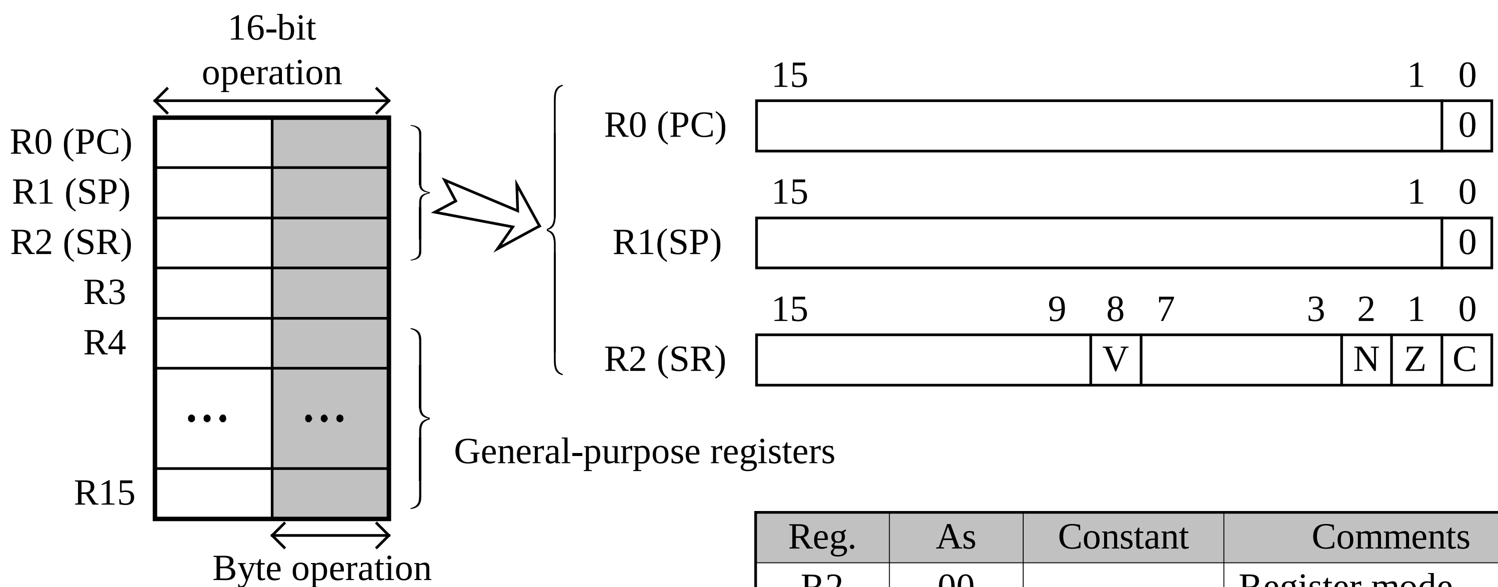


(a) CPU basic operations (b) More detailed CPU operations

The Software Model of CPU

- The software model of CPU:
  - the programming model
  - instruction formats
  - addressing modes
  - instruction set

The Programming Mode



Reg.	As	Constant	Comments
R2	00	---	Register mode
R2	01	(0)	Absolute address mode
R2	10	0x0004	+4, bit processing
R2	11	0x0008	+8, bit processing
R3	00	0x0000	0, word processing
R3	01	0x0001	+1, bit processing
R3	10	0x0010	+2, bit processing
R3	11	0xFFFF	-1, word processing

Instruction Formats

- Any instruction is composed of two major parts:
  - Opcode defines the operations of the instruction.
  - Operand specifies the operands to be operated by the instruction.

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">Opcode</td><td colspan="4">Rs</td><td>Ad</td><td>B/W</td><td colspan="2">As</td><td colspan="4">Rd</td></tr>
</table>

(a) Double-operand instruction format

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="9">Opcode</td><td>B/W</td><td colspan="2">Ad</td><td colspan="4">Rd/s</td></tr>
</table>

(b) Single-operand instruction format

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="3">Opcode</td><td colspan="3">Cond.</td><td colspan="10">10-bit offset</td></tr>
</table>

(c) Jump instruction format

Addressing Modes

- Addressing modes are the ways that operands are fetched.
  - register
  - indexed
  - register indirect
  - immediate

As/Ad	Addressing mode	Syntax	Comments
00/0	Register mode	Rn	Register contents are operand.
01/1	Indexed mode	X(Rn)	(Rn+X) points to the operand.
	Symbolic mode	ADDR	(PC+X) points to the operand.
	Absolute mode	&ADDR	X(SR) points to the operand.
10/-	Register indirect mode	@Rn	Rn points to the operand.
	Autoincrement mode	@Rn+	Rn points to the operand and increments 1 or 2.
11/-	Immediate mode	#N	@PC+.

The Instruction Set

❖ Double-operand instruction set

Mnemonic	Operation	N	Z	V	C	Op code
MOV(.B) src, dst	dst ← src	-	-	-	-	0x4xxx
ADD(.B) src, dst	dst ← src + dst	*	*	*	*	0x5xxx
ADDC(.B) src, dst	dst ← src + dst + C	*	*	*	*	0x6xxx
SUB(.B) src, dst	dst ← .not.src + dst + 1	*	*	*	*	0x8xxx
SUBC(.B) src, dst	dst ← .not.src + dst + C	*	*	*	*	0x7xxx
CMP(.B) src, dst	dst - src	*	*	*	*	0x9xxx
DADD(.B) src, dst	dst ← src + dst + C (decimal)	*	*	*	*	0xAxxx
BIT(.B) src, dst	dst .and. src	*	*	0	*	0xBxxx
BIC(.B) src, dst	dst ← .not.src .and. dst	-	-	-	-	0xCxxx
BIS(.B) src, dst	dst ← src .or. dst	-	-	-	-	0xDxxx
XOR(.B) src, dst	dst ← src .xor. dst	*	*	*	*	0xExxx
AND(.B) src, dst	dst ← src .and. dst	*	*	0	*	0xFxxx

The Instruction Set

❖ Single-operand instruction set

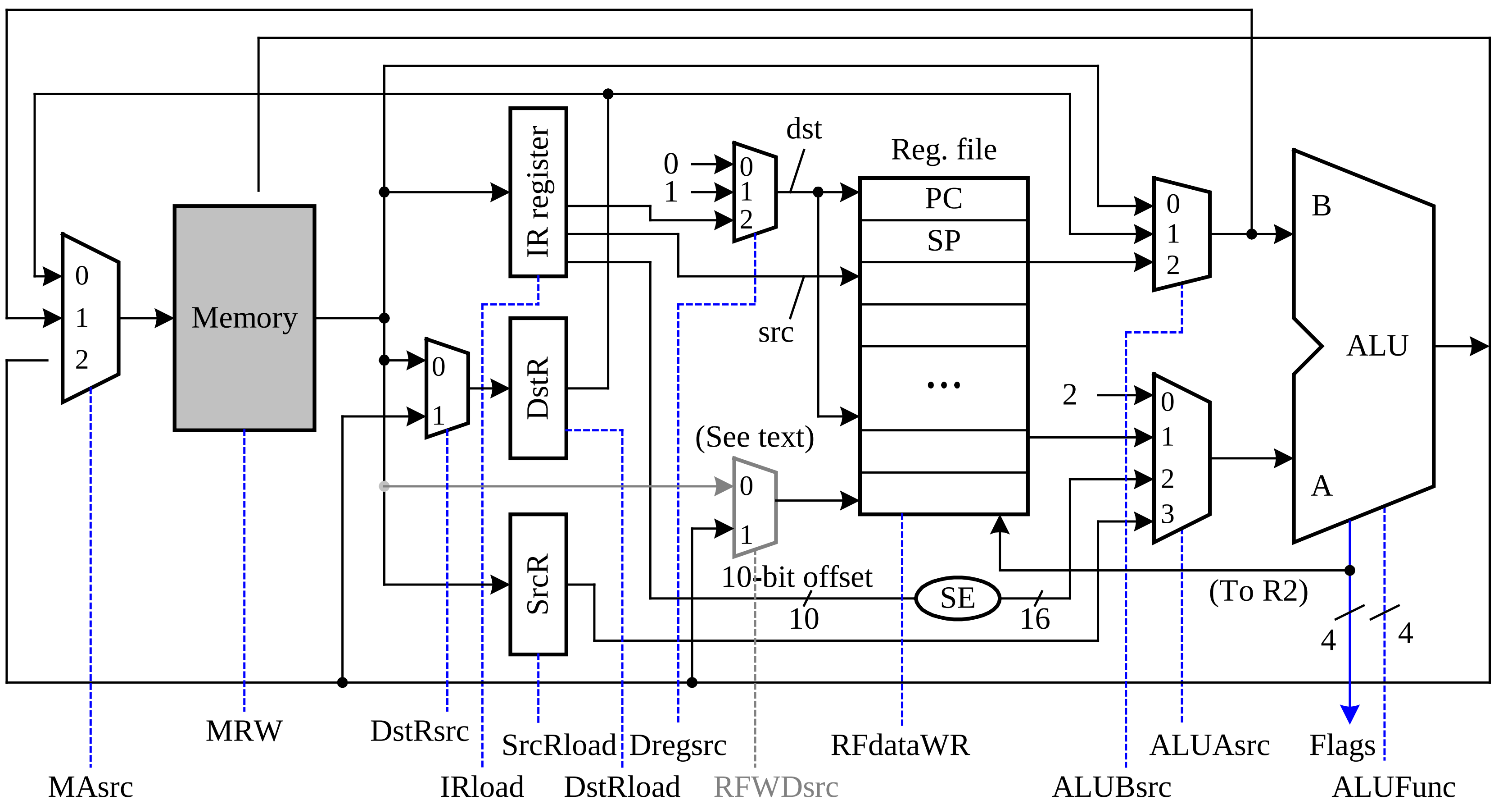
Mnemonic	Operation	N	Z	V	C	Op code
RRA(.B) dst	Arithmetic shift right, C ← LSB	*	*	0	*	0x110x
RRC(.B) dst	Rotate right through carry	*	*	*	*	0x100x
PUSH(.B) src	SP ← SP - 2, @SP ← src	-	-	-	-	0x120x
SWAPB	Swap bytes	-	-	-	-	0x108x
CALL dst	SP ← SP - 2, @SP← PC + 2, PC ← dst	-	-	-	-	0x128x
RETI	SR ← TOS, SP ← SP + 2, PC ← TOS, SP ← SP + 2,	*	*	*	*	0x130x
SXT dst	dst[15:8] ← dst[7]	*	*	0	*	0x118x

The Instruction Set

❖ Jump instruction set

Mnemonic	Operation	N	Z	V	C	Op code
JNE/JNZ label	Jump to label if zero bit is reset	-	-	-	-	0x20xx
JEQ/JZ label	Jump to label if zero bit is set	-	-	-	-	0x24xx
JC label	Jump to label if carry bit is set	-	-	-	-	0x2Cxx
JNC label	Jump to label if carry bit is reset	-	-	-	-	0x28xx
JN label	Jump to label if negative bit is set	-	-	-	-	0x30xx
JGE label	Jump to label if (N .xor. V) = 0	-	-	-	-	0x34xx
JL label	Jump to label if (N .xor. V) = 1	-	-	-	-	0x38xx
JMP label	Jump to label unconditionally	-	-	-	-	0x3Cxx

A Datapath Design

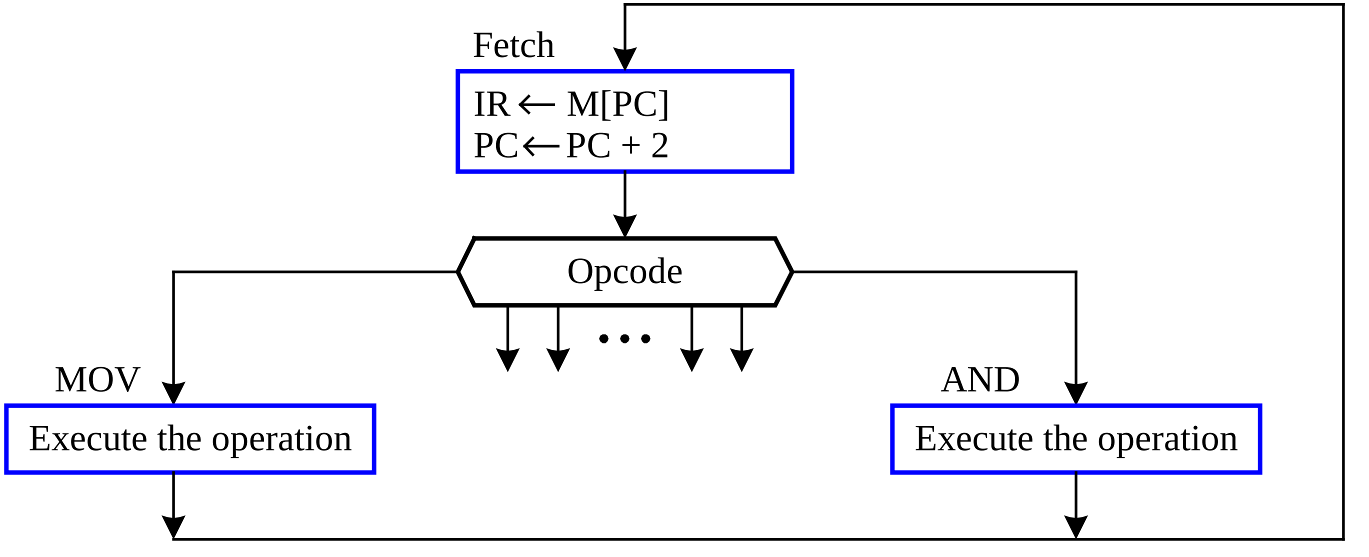


ALU Functions

Instruction	ALU function	Mode selection					Mnemonic
		m3	m2	m1	m0	B/W	
MOV(.B)	F ← A	0	0	0	0	-	pass A
MOV(.B)	F ← B	0	0	0	1	-	pass B
ADD(.B)	A+B	0	0	1	0	0/1	add(b/w)
ADDC(.B)	A+B+C	0	0	1	1	0/1	addc(b/w)
SUB(.B), CMP(.B)	A+ not B + 1	0	1	0	0	0/1	sub(b/w)
SUBC(.B)	A+ not B + not C	0	1	0	1	0/1	subc(b/w)
DADD(.B)	A+6H, A+60H conditional	0	1	1	0	0/1	dadd(b/w)
AND(.B), BIT(.B)	A and B	0	1	1	1	0/1	and(b/w)
BIC(.B)	not A and B	1	0	0	0	0/1	bic(b/w)
BIS(.B)	A or B	1	0	0	1	0/1	or(b/w)
XOR(.B)	A xor B	1	0	1	0	0/1	xor(b/w)
RRA(.B)	Arithmetic shift right into C	1	0	1	1	0/1	asrc(b/w)
RRC(.B)	Rotate right through C	1	1	0	0	0/1	rotatec(b/w)
SWAP	Swap byte	1	1	0	1	0	swap

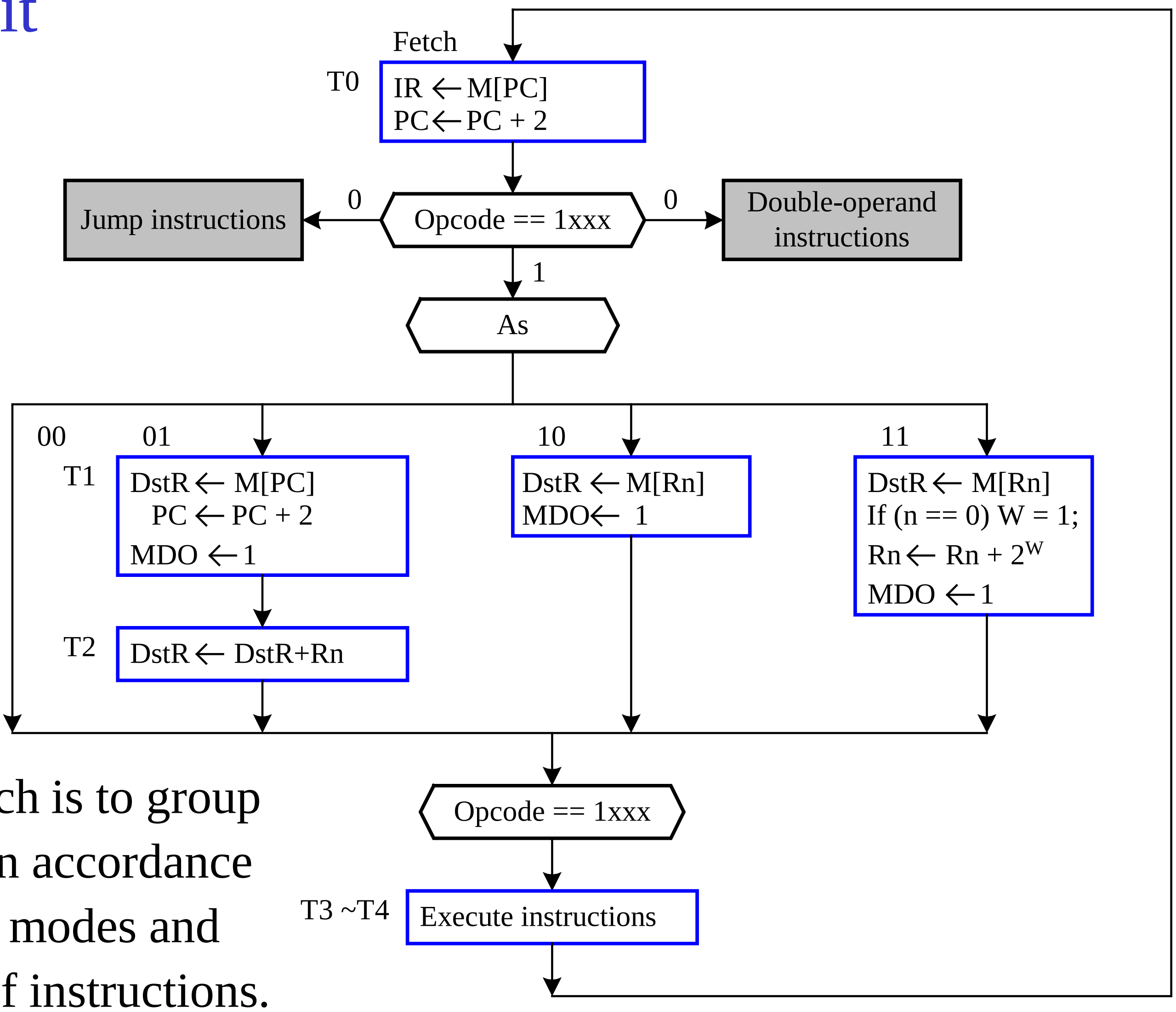
A Control Unit

- The decoder-based approach
  - An opcode decoder is used after the instruction fetch phase.
  - Each instruction is then executed accordingly.
  - The resulting states required is rather large.



A Control Unit

- Single-operand instruction



- A better approach is to group the instruction in accordance with addressing modes and the operations of instructions.

A Control Unit

- The operations of T3 and T4 are determined separately by each instruction.

Instruction	MDO	T3	T4
RRA (0001_0001_00)	MDO =0	Rd ← ARS(Rd)	
	MDO =1	SrcR ← M[DstR]	M[DstR] ← ARS(SrcR)
RRC (0001_0000_00)	MDO =0	Rd ← Rotate(Rd)	
	MDO =1	SrcR ← M[DstR]	M[DstR] ← Rotate(SrcR)
PUSH (0001_0010_00)	MDO =0	SP ← SP -2	M[SP] ← Rs
	MDO =1	SP ← SP -2 SrcR ← M[DstR]	M[SP] ← SrcR