Digital Logic Design
234262

Alex Bronstein

Lecture 10: I/O and Communication
Abstraction Hierarchy

Application Software

Operating System *(threads, files, exceptions)*

**Computer Architecture *(instruction set)***

**Micro-Architecture *(execution pipeline)***

Logic *(adders, multipliers, FSMs)*

Digital Circuits *(gates)*

Analog Circuits *(amplifiers)*

Process and Devices *(MOS FET transistors)*

Physics *(electrons, holes, tunneling, band gap)*
Memory-mapped I/O

- Certain memory addresses are mapped to device registers.
- Writing to such address sends data to external device.
- SW can poll device register to check for event.
- More efficiently: device can generate interrupt when event happens.
- Some device registers can be read-only.
Console I/O in MIPS

- **Receiver control** (0xFFFF0000)
  
  | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
  |----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

- **Receiver data** (0xFFFF0004)
  
  | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
  |----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

- **Transmitter control** (0xFFFF0008)
  
  | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
  |----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

- **Transmitter data** (0xFFFF000C)
  
  | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
  |----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
Direct keyboard input

- Receiver = keyboard
- **Receiver control register:**
  - **Ready** bit (0): asserted when unread input from keyboard
  - **Interrupt enable** bit (1): keyboard input generates interrupt
- **Receiver data register:** contains last pressed key in lowest byte
  - Reading data register resets ready bit
- Read by polling:

```assembly
li $t0, 0xffffffff

rd_poll: lw $v0, 0($t0)
andi $v0, $v0, 0x01  # key pressed?
beq $v0, $zero, rd_poll  # wait if not
lw $v0, 4($t0)  # last key in $v0
```
Direct character display

- Transmitted = display
- Transmitter control register:
  - **Ready** bit (0): asserted when display accepting new character
  - **Interrupt enable** bit (1): display ready generates interrupt
- Transmitter data register: contains last displayed character
  - Writing data register resets ready bit
- Write by polling: # character to print in $a0
  ```
  li $t0, 0xfffff0008
  wr_poll: lw $v0, 0($t0)
  andi $v0, $v0, 0x01  #display ready?
  beq $v0, $zero, wr_poll  #wait if not
  sw $a0, 4($t0)  #display character
  ```
Parallel vs Serial

**Parallel**

<table>
<thead>
<tr>
<th>TX</th>
<th>RX</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>0</td>
</tr>
<tr>
<td>D1</td>
<td>1</td>
</tr>
<tr>
<td>D2</td>
<td>1</td>
</tr>
<tr>
<td>D3</td>
<td>0</td>
</tr>
<tr>
<td>D4</td>
<td>0</td>
</tr>
<tr>
<td>D5</td>
<td>0</td>
</tr>
<tr>
<td>D6</td>
<td>1</td>
</tr>
<tr>
<td>D7</td>
<td>0</td>
</tr>
</tbody>
</table>

**Serial**

- DO
  - TX: 0 1 1 0 0 0 0 1 0
  - RX: DI

D0 D1 D2 D3 D4 D5 D6 D7
Parallel vs Serial

- **Serial**: one bit is conveyed at a time over communication link
  - Small ICs: SPI, I²C
  - Newer peripheral buses: PCIe, USB, SATA, FireWire
  - Network: Ethernet, Infiniband, Fibre channel
  - Good old serial port (RS-232)

- **Parallel**: multiple bits are sent simultaneously over the link
  - Buses inside the chip
  - Memory interfaces
  - Older peripheral buses: ISA, ATA, SCSI, PCI
  - Parallel printer port
Parallel

Pros
- Higher data rate at lower clock speed
- Simple hardware: only one register is required on each side to send/receive data on the bus

Cons
- Clock skew between channels reduces link speed
- Reduced cable length
- Increased cable/connector cost
Serial

Pros
- Clock skew is not an issue (for unclocked async links)
- Simple connectors and cables
- Extra space allows better isolation from environment
- Longer cables
- Can be clocked faster

Cons
- Requires data serialization/deserialization
Universal Asynchronous Receiver/Transmitter

- Individual or part of a bigger IC used for serial communications
- Underlying physical layer may differ (RS-232 port, USB, Bluetooth)
- Communication can be
  - **Simplex**: unidirectional
  - **Full duplex**: both devices send and receive at the same time
  - **Half duplex**: devices take turns in sending/receiving
- No clock signal is exchanged between Tx and Rx
- Requires agreement in advance between Tx and Rx on communication parameters (bps, number of data bits, stop bits, parity)
- Serialization/deserialization via **shift registers**
Shift register: serial in parallel out

Parallel out

Optional output buffer

Serial In

SHIFT

CLK

D Q

D Q

D Q

D Q
Shift register: parallel in serial out

Parallel in:
- D0
- D1
- D2
- D3

Optional input buffer

Serial Out

WR/SHIFT

CLK
UART data frame

- Line is held high (telegraph legacy)
- Frame starts with **start bit** (“0”)
- Followed by **data bits** (6-8)
- Followed by optional **parity bit** for error detection
  - **Even parity**: (data, parity) always has an even number of 1’s
  - **Odd parity**: (data, parity) always has an odd number of 1’s
  - **Fixed** at either 0 or 1
- Followed by **stop bits** (1, 1.5 or 2 at “1”)
UART Rx

- Since no clock is transmitted, Rx has to recover it from the data stream
- Done by **oversampling** (typically x16): each serial bit is sampled 16 times (but only one bit is saved)
- Rx sequence:
  1. Wait until input becomes 0 (start bit) and reset sampling counter
  2. When counter reaches 7 (middle of start bit) restart counter
  3. Wait until counter reaches 15 (middle of D0). Shift it into register and restart counter
  4. Repeat 3 for remaining data bits
  5. (Repeat 3 for parity bit)
  6. Repeat 3 for stop bits
**UART protocol**

- Sender waits until `finished` is high
- Sender writes 8 bits to `wdata` and asserts `send` to initiate transmission. Tx interface automatically resets `finished`.
- Tx interface serializes the data and sends them over the link.
- Tx interface raises `finish` when done (this can generate HW interrupt).
- Sender party writes next 8 bits...
- Rx interface samples link channel and deserializes data.
- Rx interface fills output buffer and raises `ready`.
- Receiver waits until `ready` is high and reads `rdata`.
- Receiver raises `received` to acknowledge end of reception. Rx interface automatically resets `ready`.
- Rx interface deserializes next frame...
- Transmitting and receiving parties have different clock domains
- **Asynchronous** communication between clock domains