# CSCE 4114
# bit twiddling (bashing) in C

David Andrews
dandrews@uark.edu

# Bit Twiddling in C

- Lecture will not cover common programming constructs and abstract data structures. The semantics are the same as you learned in PFI and PFII.
- If you are not familiar with C or C++ syntax many good general reviews are available online.
- We will cover the bit manipulation operations that are common and convenient for programming embedded systems.

# Bit Twiddling in C

- C was developed for writing operating systems, it is close to the machine.
  - Which OS was it created for and used to write ?
- Can Address specific memory locations
  - Can move data between memory and registers
  - Can declare register variables
- Basic data types of machine
  - Based on original Data Types of Target Mainframe
  - Needed for OS interactions with device's registers
- Operators included to manipulate single bits within these data types

# Data Types

Table 2.2.1: Data types.

| Type | Minimum size* | Range | Notes |
|---|---|---|---|
| ***signed char*** | 8 | $-128$ to $127$ | |
| ***unsigned char*** | 8 | $0$ to $255$ | |
| ***signed short*** | 16 | $-2^{15}$ to $2^{15}-1$ | $2^{16}$ is $65,536$ |
| ***unsigned short*** | 16 | $0$ to $2^{16}-1$ | |
| ***signed long*** | 32 | $-2^{31}$ to $2^{31}-1$ | $2^{32}$ is about 4 billion |
| ***unsigned long*** | 32 | $0$ to $2^{32}-1$ | |
| ***signed int*** | N | $-2^{N-1}$ to $2^{N-1}-1$ | *Though commonly used, we avoid these due to undefined width* |
| ***unsigned int*** | N | $0$ to $2^{N}-1$ | |

*The size of integer numeric data types can vary between compilers, for reasons beyond our scope. The following table lists the sizes for numeric integer data types used in this material along with the minimum size for those data types defined by the language standard.

Figure 2.2.1: Variable declarations.

```
unsigned char  ucI1;
unsigned short usI2;
signed   long  slI3;
unsigned char  bMyBitVar;
```

Feedback?

# Bit Manipulation

- C has standard bit-manipulation operators.

  & Bit-wise AND

  | Bit-wise OR          a |= 0x4; /* Set bit 2 */

  ^ Bit-wise XOR         d ^= (1 << 5); /* Toggle bit 5 */

  ~ Negate (one's comp)

  >> Right-shift         g <<= 2; /* Multiply g by 4 */

  << Left-shift          e >>= 2; /* Divide e by 4 */

What do these do ?

b &= ~0x4;

c &= ~(1 << 3);

# Bit Manipulation

- C has standard bit-manipulation operators.

& Bit-wise AND

| Bit-wise OR          a |= 0x4; /* Set bit 2 */

^ Bit-wise XOR          d ^= (1 << 5); /* Toggle bit 5 */

~ Negate (one's comp)

>> Right-shift          g <<= 2; /* Multiply g by 4 */

<< Left-shift          e >>= 2; /* Divide e by 4 */

What do these do ?

b &= ~0x4; /* Clear bit 2 */

c &= ~(1 << 3); /* Clear bit 3 */

# Example Usage

base_addr = 0x00000400

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| i/o | i/o | i/o | i/o | i/o | i/o | i/o | i/o |

- Suppose you have a Control Register that sets directions for 8 input/output devices
- 1 := input
- 0 := output
- Device is already configured, and you want to check device #2.
- Write down the code in C…………

# Example Usage

base_addr = 0x00000400

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| i/o | i/o | i/o | i/o | i/o | i/o | i/o | i/o |

- Suppose you have a Control Register that sets directions for 8 input/output devices
- 1 := input
- 0 := output
- Device is already configured, and you want to check device #2.

```
int mask = 0x4;              /* 00000100 */
A = *base_addr & mask;    /*A only has bit 2 */
```

# Ternary Operator

**if-then-else style:**

```
int a = 10, b = 20, c;

if (a < b) {
      c = a;
}
else {
      c = b;
}
```

**Ternary Operator**

```
int a = 10, b = 20, c;

c = (a < b) ? a : b
```

# Ternary Operator

Write a function using the ternary operator to set the kth bit of a word to either {0, 1}.

```
// x: 8-bit value.
//k: bit position to set, range is 0-7.
// b: set bit to this, either 1 or 0
unsigned char SetBit(
unsigned char x, unsigned char k, unsigned char b)
```

# Ternary Operator

Write a function using the ternary operator to set the kth bit of a word to either {0, 1}.

```
// x: 8-bit value.
//k: bit position to set, range is 0-7.
// b: set bit to this, either 1 or 0
unsigned char SetBit(
unsigned char x, unsigned char k, unsigned char b)

{ return (b ? (x | (0x01 << k)) : (x & ~(0x01 << k)) ); }
```

# Ternary Operator

**Write a function using the ternary operator that returns the value of the kth bit in an 8-bit word.**

// x: 8-bit value.

//k: bit position to set, range is 0-7.

# Ternary Operator

Write a function using the ternary operator that returns the value of the kth bit in an 8-bit word.

```
// x: 8-bit value.
//k: bit position to set, range is 0-7.

unsigned char GetBit(unsigned char x, unsigned char k)
{ return ((x & (0x01 << k)) != 0); }
```

# Serial/Parallel I/O

How to interface external signals/data into Computer ?

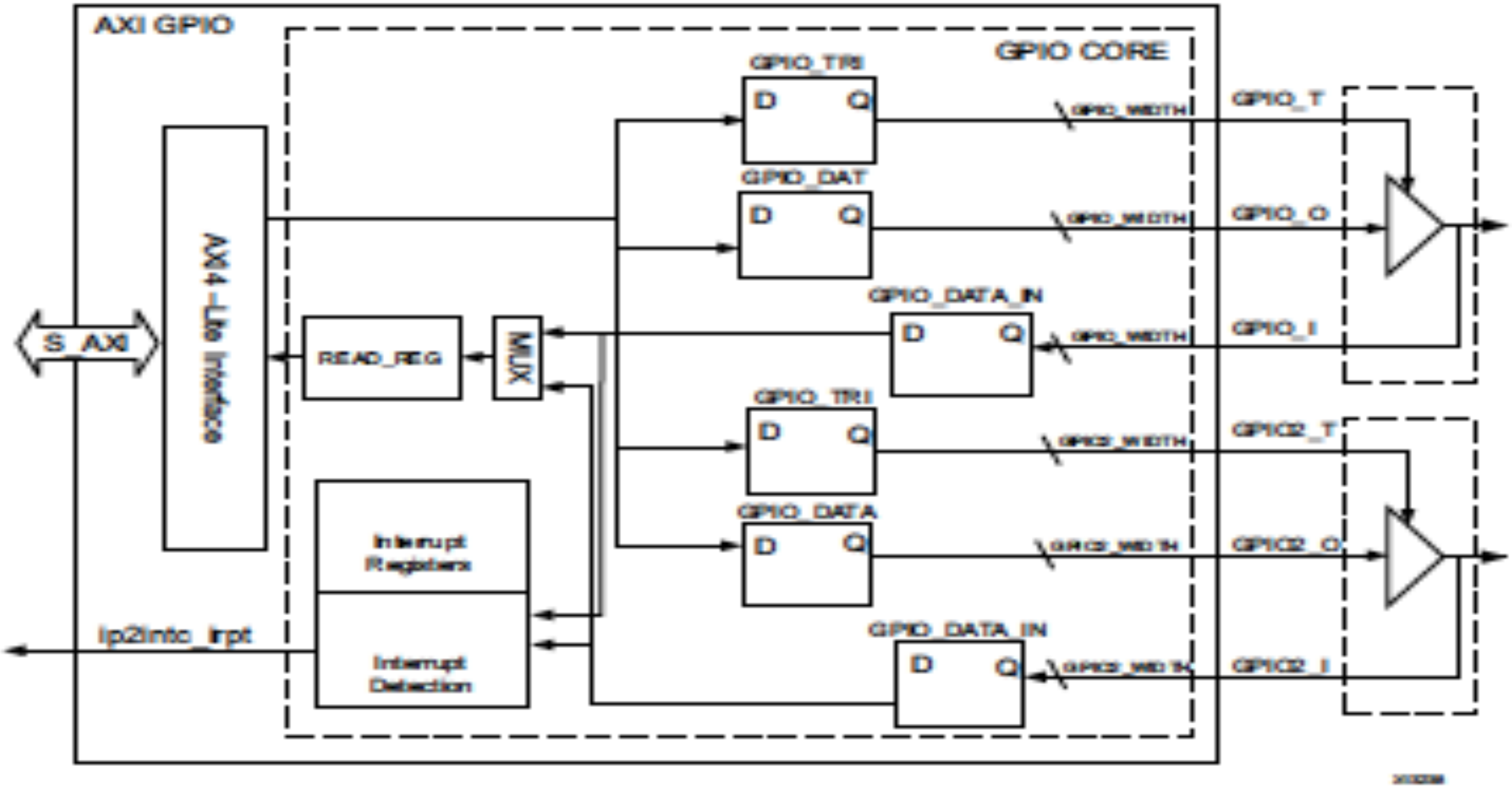
**Intel 8255 Programmable Peripheral Interface (PPI)**
**Motorola 6820 Peripheral Interface Adapter (PIA)**

Standard chips that provided general-purpose I/O lines and control lines for handshaking to external devices. Programmability allows different numbers of I/O lines to be set as either inputs or outputs depending on needs.

# GPIO: General Purpose Input/Output Core

- Provides all signals/connections to AXI bus
  - AXI (A)dvanced e(X)tensible (I)nterface
  - Part of ARM Advanced Microcontroller Bus Arch (AMBA)
- Can Have 1 or 2 Channels of 32 bits each
- Each bit can be configured as input/output or tri-state

# Schematic (Hardware Perspective)



# Registers (Programmers Perspective)

- GPIO_TRI := sets up direction and use of Tri-State
  - 0 := write (output) (also turns on tristate connections)
  - 1 := read (input) (disables tristate connections)
    - Tri-state or dedicated input/output pins set during system build

*Table 2-4:* **Registers**

| Address Space Offset[3] | Register Name | Access Type | Default Value | Description |
|---|---|---|---|---|
| 0x0000 | GPIO_DATA | R/W | 0x0 | Channel 1 AXI GPIO Data Register. |
| 0x0004 | GPIO_TRI | R/W | 0x0 | Channel 1 AXI GPIO 3-state Control Register. |
| 0x0008 | GPIO2_DATA | R/W | 0x0 | Channel 2 AXI GPIO Data Register. |
| 0x000C | GPIO2_TRI | R/W | 0x0 | Channel 2 AXI GPIO 3-state Control. |
| 0x011C | GIER[1] | R/W | 0x0 | Global Interrupt Enable Register. |
| 0x0128 | IP IER[1] | R/W | 0x0 | IP Interrupt Enable Register (IP IER). |
| 0x0120 | IP ISR[1] | R/TOW[2] | 0x0 | IP Interrupt Status Register. |

# Data Port

- GPIOx_Data := Port for Data
  - If a bit configured as Output:
    - Writing to it will output the data
    - Bit cannot be read
  - If a bit configured as Input
    - Reading will bring in value
    - Writing to it won't do anything

# Example Code Use

```
/* Push buttons are used to control the on-board LEDs. */
// Direction Masks
#define outputDir   0x00000000    // All output bits
#define inputDir     0x0000001F    // 5-input bits

int main()
{
  // Pointer defintions for Button GPIO
  //  ** NOTE - integer definition causes
  //    offsets to be automatically be multiplied by 4!!
  volatile int *base_buttonGPIO          = (int*)(0x40040000);
  volatile int *data_buttonGPIO          = (int*)(base_buttonGPIO + 0x0);
  volatile int *tri_buttonGPIO           = (int*)(base_buttonGPIO + 0x1);

  // Pointer defintions for LED GPIO
  //  ** NOTE - integer definition causes
  //      offsets to be automatically  be multiplied by 4!!
  volatile int *base_ledGPIO             = (int*)(0x40000000);
  volatile int *data_ledGPIO             = (int*)(base_ledGPIO + 0x0);
  volatile int *tri_ledGPIO              = (int*)(base_ledGPIO + 0x1);
```

```
// Variable used to store the state of the buttons
  int data = 0;

// Init. the LED peripheral to outputs
  print("Init. LED GPIO Data Direction...\r\n");
  *tri_ledGPIO = outputDir;

  // Init. the Button peripheral to inputs
  print("Init. Button GPIO Data Direction...\r\n");
  *tri_buttonGPIO = inputDir;

  // Infinitely Loop...
  while(1)
   {  // Read the current state of the push buttons
       data = *data_buttonGPIO;
       xil_printf("buttonState = %d\r\n",data);

       // Set the state of the LEDs
       *data_ledGPIO = data; }
  return 0;
}
```