Memory mapped IO vs IO mapped IO

- M: IO devices (SFRs) are mapped as memory addresses (in memory addr space).
- I: IO devices (SFRs) are mapped as seperate IO addresses (in diff addr space).
- M: IO regrs are accessed with same instructions as of memory e.g. MOV, LD/ST.
- I: IO regrs are accessed with special instructions e.g. IN/OUT.
- M: There is no differentiation betn mem locations and IO regrs -- except addrs.
- I: Mem locs and IO regrs are differentiated by different buses or control signal e.g. IO/M'
- M: Example: ARM.
- I: Example: x86 arch.

RISC vs CISC

- High level languages --> Compiler --> Low level languages
- CISC arch --> Complex instruction set
  - High level language constructs (if-else, loops, ...) --> Assembly language
  - Simplified compiler code generation
  - Complex instructions (macro-instructions) --> While execution, divided into micro-instructions
  - Complex execution engine --> More number logic gates
- RISC arch --> Reduced instruction set
  - Minimal instruction set (micro-instructions).
  - Simplified execution engine --> Less number logic gates
  - Complex compiler code generation

RISC vs CISC

- R: Reduced Instruction Set Computing
- C: Complex Instruction Set Computing
- R: less number of instructions
- C: more number of instructions
- R: all instructions are micro-instructions -- cpu doesn't further divide it into smaller units.
- C: some instructions are macro-instructions -- cpu further divides it into smaller units and then execute.
- R: most of the instructions are executed in single cpu cycle.
- C: needs variable number of cycles from 1-6.
- R: instruction width is fixed i.e. most of the instructions are of same size.
- C: instruction width is variable.
- R: large number of general purpose registers.
- C: less number of general purpose registers.
- R: follows load/store pattern i.e. data must be loaded into CPU regrs from RAM before manipulating it.
- C: some instructions can directly manipulate data present in main memory.
- R: compiler design more complicated; but more efficient execution.
- C: Becauze of macro-instructions, compiler design (for high level language) is easier.
- R: typically uses instruction pipeline -- instruction level parallelism.
- C: typically uses instruction queue -- fetch next instruction.
- R: Examples - AVR, ARM, ...
- C: Examples - 8086, 8051, ...

Instruction Pipeline vs Instruction Queue

- Q: Temp storage for the instruction before it is executed.
- P: Multiple independent but cascaded processing stages process instructions simultaneously.
- Q: instruction is only stored not processed.
- P: each stage is doing some processing on instruction.
- Q: typically used in CISC arch like x86.
- P: typically used in RISC arch like AVR, PIC, ARM, ..

Instruction Pipeline

- Instruction pipeline contains multiple independent units that process instruction partially. All these units can execute simultaneously and thus multiple instructions can be processed parallely. This is called as "instruction level parallelism".
- Example:
  - 2-stage: Fetch --> Execute
    - AVR & PIC
  - 3-stage: Fetch --> Decode --> Execute
    - ARM7 & ARM-CM3

Pipeline Problems/Hazards

Control Hazards

- Due to jump or conditional jump, instructions already fetched into pipeline are of no use and hence pipeline need to be cleared and re-filled from new (jumped) address. Obviously next instruction is not effectively completed in single CPU cycle.
- To overcome this problem, CPU can use "branch prediction" i.e. CPU predicts whether condition in current jump instruction will be true or not. Depending on that it fetch the next instructions.
- Some advanced CPUs also use "brach speculation". In this next instructions are not only fetched but also processed and their result is stored temporarily. If prediction is correct, the result is utilized (for faster execution); otherwise the result is discarded.

Data Hazards

- For some instructions more CPU cycles are needed for the execution eg. mul. If next instruction execution depends on result of previous instruction, then pipeline is stalled (paused) until current instruction is completed. Due to this effective number of CPU cycles for execution of instruction will increase.

Structural Hazards

- Due to CPU design restrictions some instructions cannot be completed immediately. e.g. If CPU is designed to write only one result into regr and two results were produced in an instruction. In this case, pipeline is stalled to complete the current instruction.

Bitwise

Helper macro (Bit Value)

```
#define BV(n)   (1 << (n))
```

check nth bit of register

```
if(regr & BV(n))
    // nth bit is 1
else
    // nth bit is 0
```

```
regr    :   0x4A    :   0100 1010
            BV(3)   :   0000 1000
            ----------------------
                        0000 1000

regr    :   0x4A    :   0100 1010
            BV(4)   :   0001 0000
            ----------------------
                        0000 0000
```

set nth bit of register

```
regr = regr | BV(n)
// or
regr |= BV(n)
```

```
regr    :       0x4A    :   0100 1010
            |   BV(3)   :   0000 1000
            -------------------------
                            0100 1010

regr    :       0x4A    :   0100 1010
            |   BV(4)   :   0001 0000
            -------------------------
                            0101 1010
```

clear nth bit of register

```
regr = regr & ~BV(n)
// or
regr &= ~BV(n)
regr    :       0x4A    :   0100 1010
                BV(3)   :   0000 1000
            &   ~BV(3)  :   1111 0111
            -------------------------
                            0100 0010

regr    :       0x4A    :   0100 1010
                BV(4)   :   0001 0000
            &   ~BV(4)  :   1110 1111
            -------------------------
                            0100 1010
```

toggle nth bit of register

```
regr = regr ^ BV(n)
// or
regr ^= BV(n)
```

```
regr    :       0x4A    :   0100 1010
            ^   BV(3)   :   0000 1000
            --------------------------
                            0100 0010

regr    :       0x4A    :   0100 1010
            ^   BV(4)   :   0001 0000
            --------------------------
                            0101 1010
```

set bits 12 to 15 of register

```
regr = regr | BV(12) | BV(13) | BV(14) | BV(15)
// or
regr |= BV(12) | BV(13) | BV(14) | BV(15)
```

```
BV(12)  :   0000 0000 0000 0000 0001 0000 0000 0000
BV(13)  :   0000 0000 0000 0000 0010 0000 0000 0000
BV(14)  :   0000 0000 0000 0000 0100 0000 0000 0000
BV(15)  :   0000 0000 0000 0000 1000 0000 0000 0000
            -----------------------------------------
            0000 0000 0000 0000 1111 0000 0000 0000
```

```
regr    :   0x00004A00  :   0000 0000 0000 0000 0100 1010 0000 0000
        |                   0000 0000 0000 0000 1111 0000 0000 0000
                ----------------------------------------------------
                            0000 0000 0000 0000 1111 1010 0000 0000
```

clear bits 17 to 20 of register

```
regr = regr & ~(BV(17) | BV(18) | BV(19) | BV(20))
// or
regr &= ~(BV(17) | BV(18) | BV(19) | BV(20))
```

```
BV(17)  :   0000 0000 0000 0010 0000 0000 0000 0000
BV(18)  :   0000 0000 0000 0100 0000 0000 0000 0000
BV(19)  :   0000 0000 0000 1000 0000 0000 0000 0000
BV(20)  :   0000 0000 0001 0000 0000 0000 0000 0000
----------------------------------------------------
|       :   0000 0000 0001 1110 0000 0000 0000 0000
----------------------------------------------------
~       :   1111 1111 1110 0001 1111 1111 1111 1111

regr    :   0x004A0000  :   0000 0000 0100 1010 0000 0000 0000 0000
                        &   1111 1111 1110 0001 1111 1111 1111 1111
                        --------------------------------------------
                            0000 0000 0100 0000 0000 0000 0000 0000
```

read value from bit 19 to 24 of register

```
mask = 0x3F << 19;              //0x01F80000
value = (regr & mask) >> 19;
```

```
0x3F    :   0011 1111

regr    :   0x004A0000  :   0000 0000 0100 1010 0000 0000 0000 0000
                            0000 0001 1111 1000 0000 0000 0000 0000
                            ----------------------------------------
                            0000 0000 0100 1000 0000 0000 0000 0000
                >>19    :   0000 0000 0000 0000 0000 0000 0000 1001
```

write value on bit 8 to 15 of register

```
value = 0x4A
regr &= ~(BV(8) | BV(9) | BV(10) | BV(11) | BV(12) | BV(13) | BV(14) | BV(15));
regr = regr | (value << 8);
// or
regr |= (value << 8);
```

```
regr    :   0x0004A000   :   0000 0000 0000 0100 1010 0000 0000 0000
        clear bit 8-15   :   0000 0000 0000 0100 0000 0000 0000 0000
value   :   0x4A         :   0000 0000 0000 0000 0000 0000 0100 1010
            << 8         :   0000 0000 0000 0000 0100 1010 0000 0000
            --------------------------------------------------------
            |            :   0000 0000 0000 0100 0100 1010 0000 0000
```

wait while bit 4 of register is 0

```
    while(regr & BV(4) == 0)
        ;
```

wait while bit 4 of register is 1

```
    while(regr & BV(4) != 0)
        ;
```

Homework

```
unsigned int swap_bits(unsigned int value, char m, char n)
{
    unsigned int res = swap mth and nth bit
    return res;
}
int main(void)
{
    num = ?
    printf("", num);
    num = swap_bits(num, 13, 20);
    printf("", num);
}
```

STM32F407VG GPIO

- Multiplexed GPIO pins
  - GPIOA - GPIOI ports
  - Max 16 IO pins per port
  - Total 82 GPIO pins
- Input / Output types
  - push-pull or open drain
  - pull-up or pull-down registors
  - floating or analog input
- Configurable speed
- External interrupt
- GPIO registers
  - MODER: Input (0), Output (1), Analog (3) or Alt (2)
  - OTYPER: Push-pull (0) or Open-drain (1)
  - PUPDR: Pull-up (1), Pull-down (2) or none (0)
  - OSPEEDR: Low (0) to Very High (3)
  - IDR: Input
  - ODR: Output
  - BSRR: Bit Set/Reset
- GPIO programming steps
  - GPIO as output
    - Enable GPIO clock (AHB1ENR)
    - Select GPIO mode as output
    - Select GPIO speed
    - Set pull-up or pull-down resistor
    - Set output type (push-pull or open-drain)
    - Set or Clear pin

LED

- 4 user LEDs are connected to PORTD
  - Pin 12 - Green
  - Pin 13 - Orange
  - Pin 14 - Red
  - Pin 15 - Blue
- All GPIO ports are connected to AHB1 bus
- To set GPIO Pin in input/output mode
  - MODER
    - 00 - input mode
    - 01 - output mode
  - 0 - 31, 29, 27, 25
  - 1 - 30, 28, 26, 24
- To set type of output (Push-Pull/Open drain)
  - OTYPER
    - 0 - push pull
    - 1 - open drain
  - 0 - 12, 13, 14, 15
- To set speed of GPIO pin

  - OSPEEDR
    - 00 - low speed
    - 01 - medium speed
    - 10 - High speed
    - 11 - very high speed
  - 0 - 24 to 31
- To activate pull up or pull down register
  - PUPDR
    - 00 - no pull up/ pull down
    - 01 - pull up
    - 10 - pull down
  - 0 - 24 to 31
- To read input from GPIO pin
  - IDR
- To write output on GPIO pin
  - ODR
  - bit 12 to 15 need to modify for LEDs
- To enable clock for GPIOD
  - AHB1ENR
    - 0 - disabled
    - 1- enabled
  - 1 - 3 bit