Day 12
More Programmable Logic
Verilog Expectations

- Introduced this mainly because this is how a lot of modern digital design is done
- Also could be very useful for some more advanced projects
- Don’t expect you to write Verilog from scratch
- Do believe you should be able to understand what simple Verilog code is doing
Example 1

module mymodule(
    input clock,
    input d,
    output [7:0] q
);

// Be clear these are registers
reg [7:0] q;

always @ (posedge clock) begin
    q <= {q[6:0], d};
end
endmodule

What kind of device is this?
```verilog
module mymodule(
    input clock,
    input d,
    output [7:0] q
);

    // Be clear these are registers
    reg [7:0] q;

    always @ (posedge clock) begin
        q <= {q[6:0], d};
    end

endmodule
```

---

This is a shift register

|------|------|------|------|------|------|------|------|------|
Example 2

module mymodule(
    input clock,
    output q
);
// A wire is an internal connection
wire q0, q1, q2;
// Create 3 flip-flops
    dff ff0(clock, !q0, q0);
    dff ff1(q0, !q1, q1);
    dff ff2(q1, !q2, q2);
    assign q = q2;
endmodule

// D-type flip flop
module dff(
    input clock,
    input d,
    output q
);
    reg q;
    always @ (posedge clock)
        q <= d;
endmodule
Example 2

```verilog
module mymodule(
    input clock,
    output q
);
    // A wire is an internal connection
    wire q0, q1, q2;
    // Create 3 flip-flops
    dff ff0(clock, !q0, q0);
    dff ff1(q0, !q1, q1);
    dff ff2(q1, !q2, q2);
    assign q = q2;
endmodule

// D-type flip flop
module dff(
    input clock,
    input d,
    output q
);
    reg q;
    always @ (posedge clock)
        q <= d;
endmodule
```

The 3 flip-flops form a ripple counter
So the output q is clock / 8
Tristate Buffers
PIC Architecture

Program Memory (ROM)
- Program Counter
  - 0x000
  - 0xFFF
- Instruction Decode
  - 12 bits
  - 14 bits

Instruction Logic
- W Reg
  - 8 bits

Data Memory (RAM)
- 0x00
- 0x7F
- 8 bits
PIC Architecture

Program Memory (ROM)

Program Counter

0x000

12

0xFFF

14 bits

14

Instruction Decode

Instruction Logic

W Reg

8

Each instruction cycle takes 4 clock cycles (~20 MHz)
PIC Architecture

1) Increment program counter
PIC Architecture

1) Load instruction into register

2) Load instruction into register
PIC Architecture

Program Memory (ROM)

Program Counter

0x000

0xFFF

12

14

Instruction Decode

W Reg

Instruction Logic

Data Memory (RAM)

0x00

0x7F

8 bits

3) Evaluate instruction, write result to W reg or data memory
PIC Architecture

4) Optionally change program counter (goto, return, jump)
• Program is a list of numbers (instructions) in ROM

• Usually don’t think about instructions directly, but with small µProcessors, it is actually quite useful
35 total instructions
Opcode fits into ~6 bits

8 bits remaining
in instruction word
for an address
or data (but not both)

<table>
<thead>
<tr>
<th>Mnemonic, Operands</th>
<th>Description</th>
<th>Cycles</th>
<th>14-Bit Instruction Word</th>
<th>Status Affected</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>f, d</td>
<td>Add W and f</td>
<td>1</td>
<td>00 0111 dfff ffff</td>
<td>C,DC,Z</td>
<td>1,2</td>
</tr>
<tr>
<td>f, d</td>
<td>AND W with f</td>
<td>1</td>
<td>00 0101 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>f</td>
<td>Clear f</td>
<td>1</td>
<td>00 0001 lfff ffff</td>
<td>Z</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>Clear W</td>
<td>1</td>
<td>00 0001 0xxx xxxx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f, d</td>
<td>Complement f</td>
<td>1</td>
<td>00 1001 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>f, d</td>
<td>Decrement f</td>
<td>1</td>
<td>00 0011 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>f, d</td>
<td>Decrement f, Skip if 0</td>
<td>1(2)</td>
<td>00 1011 dfff ffff</td>
<td>Z</td>
<td>1,2,3</td>
</tr>
<tr>
<td>f, d</td>
<td>Increment f</td>
<td>1</td>
<td>00 1010 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>f, d</td>
<td>Increment f, Skip if 0</td>
<td>1(2)</td>
<td>00 1111 dfff ffff</td>
<td>Z</td>
<td>1,2,3</td>
</tr>
<tr>
<td>f, d</td>
<td>Inclusive OR W with f</td>
<td>1</td>
<td>00 0100 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>f, d</td>
<td>Move f</td>
<td>1</td>
<td>00 1000 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>f, d</td>
<td>Move W to f</td>
<td>1</td>
<td>00 0000 lfff ffff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>No Operation</td>
<td>1</td>
<td>00 0000 0xx0 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f, d</td>
<td>Rotate Left f through Carry</td>
<td>1</td>
<td>00 1101 dfff ffff</td>
<td>C</td>
<td>1,2</td>
</tr>
<tr>
<td>f, d</td>
<td>Rotate Right f through Carry</td>
<td>1</td>
<td>00 1100 dfff ffff</td>
<td>C</td>
<td>1,2</td>
</tr>
<tr>
<td>f, d</td>
<td>Subtract W from f</td>
<td>1</td>
<td>00 0010 dfff ffff</td>
<td>C,DC,Z</td>
<td>1,2</td>
</tr>
<tr>
<td>f, d</td>
<td>Swap nibbles in f</td>
<td>1</td>
<td>00 1110 dfff ffff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f, d</td>
<td>Exclusive OR W with f</td>
<td>1</td>
<td>00 0110 dfff ffff</td>
<td>Z</td>
<td>1,2</td>
</tr>
<tr>
<td>f, b</td>
<td>Bit Clear f</td>
<td>1</td>
<td>01 00bb bfff ffff</td>
<td></td>
<td>1,2</td>
</tr>
<tr>
<td>f, b</td>
<td>Bit Set f</td>
<td>1</td>
<td>01 01bb bfff ffff</td>
<td></td>
<td>1,2</td>
</tr>
<tr>
<td>f, b</td>
<td>Bit Test f, Skip if Clear</td>
<td>1 (2)</td>
<td>01 10bb bfff ffff</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>f, b</td>
<td>Bit Test f, Skip if Set</td>
<td>1 (2)</td>
<td>01 11bb bfff ffff</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>k</td>
<td>Add literal and W</td>
<td>1</td>
<td>11 111x kkkk kkkk</td>
<td>C,DC,Z</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>AND literal with W</td>
<td>1</td>
<td>11 1001 kkkk kkkk</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>Call subroutine</td>
<td>2</td>
<td>10 0kkk kkkk kkkk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Clear Watchdog Timer</td>
<td>1</td>
<td>00 0000 0110 0100</td>
<td>TO,PD</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>Go to address</td>
<td>2</td>
<td>10 1kkk kkkk kkkk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>Inclusive OR literal with W</td>
<td>1</td>
<td>11 1000 kkkk kkkk</td>
<td>Z</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>Move literal to W</td>
<td>1</td>
<td>11 00xx kkkk kkkk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Return from interrupt</td>
<td>2</td>
<td>00 0000 0000 1001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>Return with literal in W</td>
<td>2</td>
<td>11 01xx kkkk kkkk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Return from Subroutine</td>
<td>2</td>
<td>00 0000 0000 1000</td>
<td>TO,PD</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>Go into standby mode</td>
<td>1</td>
<td>00 0000 0110 0011</td>
<td>TO,PD</td>
<td>C,DC,Z</td>
</tr>
<tr>
<td>k</td>
<td>Subtract W from literal</td>
<td>1</td>
<td>11 110x kkkk kkkk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>Exclusive OR literal with W</td>
<td>1</td>
<td>11 1010 kkkk kkkk</td>
<td>Z</td>
<td></td>
</tr>
</tbody>
</table>
Example Instruction

<table>
<thead>
<tr>
<th>Mnemonic, Operands</th>
<th>Description</th>
<th>Cycles</th>
<th>14-Bit Instruction Word</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSb</td>
</tr>
<tr>
<td><strong>BYTE-ORIENTED FILE REGISTER OPERATIONS</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDWF f, d</td>
<td>Add W and f</td>
<td>1</td>
<td>00 0111 dfff ffff</td>
</tr>
<tr>
<td>ANDWF f, d</td>
<td>AND W with f</td>
<td>1</td>
<td>00 0110 dfff ffff</td>
</tr>
<tr>
<td><strong>CLRF f</strong></td>
<td>Clear f</td>
<td>1</td>
<td>00 0001 1fff ffff</td>
</tr>
<tr>
<td><strong>CLRW</strong></td>
<td>Clear W</td>
<td>1</td>
<td>00 0001 0xxx xxxx</td>
</tr>
</tbody>
</table>

Assembly Code:

CLRF f ; Clear (fill with 0x00) memory at address f

**Byte-oriented** file register operations

<table>
<thead>
<tr>
<th>13</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OPCODE</strong></td>
<td>d</td>
<td>f (FILE #)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

d = 0 for destination W
d = 1 for destination f
f = 7-bit file register address
Example Assembly Code

CountLoop:

  incf  COUNT, w         ; increment COUNT and store result in W
  andlw 0x07             ; mask off three lowest bits in W
  movwf  GPIO            ; copy result to output register
  movwf  COUNT           ; copy result back to COUNT
  call   WaitSub         ; Call sub-routine to wait for a while
                       ; Execution returns here when done
  goto   CountLoop       ; Go back to start of loop

Each line is one instruction
Often use symbolic rather than literal addresses
Readable (if you know what you are looking for)
Assembler turns this into a string of numbers
Adding 2 numbers

Can’t specify 2 addresses in one command...

MOVF f, 0 ; Move number in memory address f to w register
; The 0 is the destination of the operation output
; Can be either w reg (0) or back in f (1)

ADDWF g, 0 ; Add value in address g to w register
; The 0 says to put the result in w reg

MOVWF, h ; Move the current value of the w reg back to h

Programming in assembly is always a bit of a dance
Only 1 ‘working’ register, but enough for any logic operation
Add 5 to a number

MOVLW 0x05 ; Move number 5 (0x05) to w register

ADDWF f, 1 ; Add value in address f to w register
; The 1 says to put the result back in address f

Operations using 8-bit literals must involve w register
Not enough bits to specify a memory address as well
### Conditional Operations

#### Bit-oriented File Register Operations

<table>
<thead>
<tr>
<th>Mnemonic, Operands</th>
<th>Description</th>
<th>Cycles</th>
<th>14-Bit Instruction Word</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>MSb</td>
</tr>
<tr>
<td>BCF</td>
<td>Bit Clear f</td>
<td>1</td>
<td>01 00bb bfff ffff</td>
</tr>
<tr>
<td>BSF</td>
<td>Bit Set f</td>
<td>1</td>
<td>01 01bb bfff ffff</td>
</tr>
<tr>
<td>BTFSC</td>
<td>Bit Test f, Skip if Clear</td>
<td>1 (2)</td>
<td>01 10bb bfff ffff</td>
</tr>
<tr>
<td>BTFSS</td>
<td>Bit Test f, Skip if Set</td>
<td>1 (2)</td>
<td>01 11bb bfff ffff</td>
</tr>
</tbody>
</table>

### Bit Test f, Skip if Set

```c
#include <stdio.h>

#define COUNT 0x20

void Counting()
{
    incf COUNT, 1;  // Increment (add 1) to count
    btfsc COUNT, 7; // Test bit 7 (MSB) of value at address COUNT
    goto Counting;  // If bit not set, execution continues here
    return;         // If bit is set, execution continues here
}
```

- Include a header file to use preprocessor directives.
- Use `#define` to define a constant for memory location.
- Use `incf` to increment the counter by 1.
- Use `btfsc` to test the most significant bit (MSB) of a value.
- Use `goto` to jump to a label.
- Use `return` to return to the caller.

This example demonstrates how to implement a simple counter with conditional branching based on a bit test.

---

Note: The example code uses preprocessor directives (`#define`, `#include`) and conditional branching (`if`, `goto`) to illustrate the use of conditional operations. The 14-bit instruction word format is for reference and may not directly correspond to the actual assembly instructions used in the code.
1.0 DEVICE OVERVIEW

This document contains device specific information for the PIC12F629/675. Additional information may be found in the PICmicro™ Mid-Range Reference Manual (DS33023), which may be obtained from your local Microchip Sales Representative or downloaded from the Microchip web site. The Reference Manual should be considered a complementary document to this Data Sheet, and is highly recommended reading for a better understanding of the device architecture and operation of the peripheral modules.

The PIC12F629 and PIC12F675 devices are covered by this Data Sheet. They are identical, except the PIC12F675 has a 10-bit A/D converter. They come in 8-pin PDIP, SOIC, and MLF-S packages. Figure 1-1 shows a block diagram of the PIC12F629/675 devices. Table 1-1 shows the Pinout Description.

![Actual PIC 12f675 diagram](image)

This is an 8-pin DIP package.
μProcessor

• Bare microprocessors aren’t actually that useful

• Modern microprocessors implement a lot of additional functionality
  • ADCs, counters, DACs, serial interfaces
  • Mapped to specific memory locations (access by reading/writing specific f values)

• Have ‘interrupt’ functionality (code execution jumps to special place if something happens, like pin value changes)

• Can be used to put chip to sleep and wake on interrupt (low power)
Memory Map

- GPIO (0x05) connects to the 6 I/O pins
- TRISIO (0x85) sets tristate status for pins
- CMCON (0x19) is an analog comparator
- ADCON0 and ANSEL configure ADC
- ADRESH/ADRESL are ADC results (10 bit ADC, takes 2 memory locations, read only)
- EEDAT/EEADR accesses internal EEPROM
µProc Summary

- Most people will program these in C or python or some other high-level language.

- Assembly programming gives you a more intimate understanding of the bit-level operations.
  - Precision timing possible (know exact cycle counts)
  - Doesn’t scale well to complex code, larger devices

- Both have their place

- Some things are easy in assembly (bit-level operations)

- Other things are very hard (division….)