Introduction to Assembly language

USING THE PIC18 MICROPROCESSOR

Outline

- Introduction to Assembly Code
- The microchip PIC18 Microprocessor
- Binary/Hex Numbers
- Breaking down an example microprocessor program
- PIC instructions overview
- Downloading and compiling assembly code
- Running and debugging assembly code

A really simple program

- A variable (i)
- Set variable to an initial value
- A loop
  - Check a condition to see if we are finished
  - Output some information
  - Increment the loop counter

```
main()
{
    char i;
    i=0;
    while (i<100) {
        PORTC = i;
        i=i+1;
    }
}
```

Running a program on a microprocessor

- When you want to turn your program into something which runs on a microprocessor you typically *compile* the program
- This creates a program in the “machine language” of the microprocessor
  - A limited set of low level instructions which can be executed directly by the microprocessor
- We can also program in this language using an *assembler*
- We can have a look at the assembly language the c compiler generates for our simple program to understand how this works

Translating our program into assembly language


```
main()
{
    char i;
    i=0;
    while (i<100) {
        PORTC = i;
        i=i+1;
    }
}
```

```
0100   0E00      MOVLW 0x0
0102   D003      BRA 0x10A
0104   C006      MOVFF 0x06, PORTC
0106   FF82      NOP
0108   2806      INCF 0x06, W, ACCESS
010A   6E06      MOVWF 0x06, ACCESS
010C   0E63      MOVLW 0x63
010E   6406      CPFSGT 0x06, ACCESS
0110   D7F9      BRA 0x104
0112   EF00      GOTO 0x0
0114   F000      NOP
```

Location in program memory (byte address)

Opcodes – the program code

The Assembly language equivalent of the opcodes

PIC18 Microprocessor architecture

Program Memory
Your code goes here

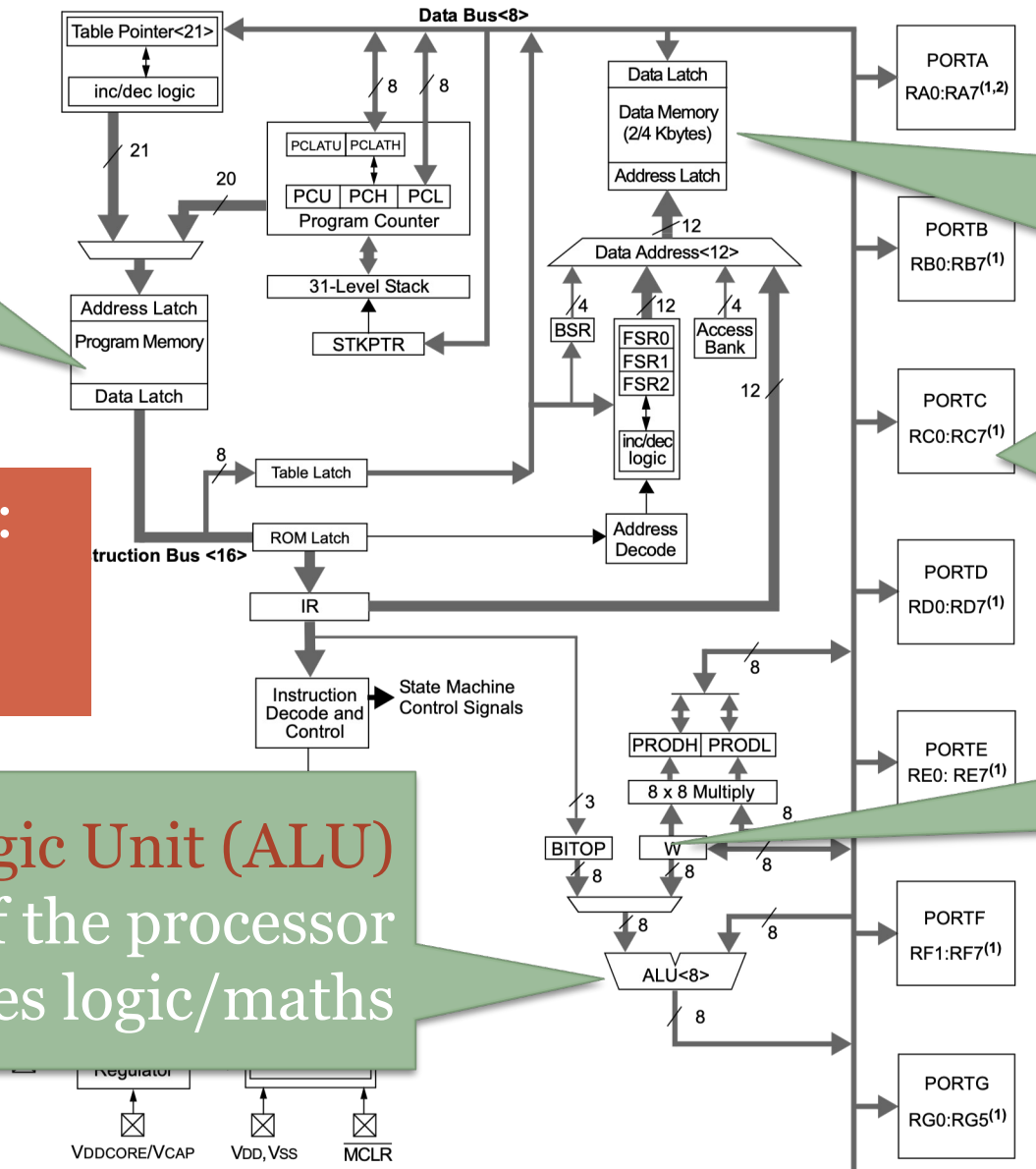
Harvard Architecture:
Program and data stored separately

Arithmetic Logic Unit (ALU)
The heart of the processor which handles logic/maths

Data memory
Storage for numbers the processer will perform arithmetic on

Input/Output
Interface to the outside world

W
Working Register



Numbers on a microprocessor

- We've seen that our program is converted into a series of numbers for execution on the microprocessor
- Numbers are stored in a microprocessor in memory
  - They can be moved in an out of memory
  - Calculations can be done
  - Numbers can be sent to Output devices and read from Input devices
- The numbers are stored internally in binary representations as `0`s and `1`s.

Binary/Hexadecimal Numbers

- A "Bit" can be a `0` or `1`
  - The value is set using transistors in the hardware
- Bits are organized into groups to represent numbers
  - 4 Bits is a "Nibble"
    - Can store numbers 0-15
  - 8 Bits is a "Byte"
    - Can store numbers 0-255
  - 16 Bits is a word
    - Can store numbers 0-65535
- Hexadecimal is a very handy representation for binary numbers
  - Each 4 bits maps onto a HEX number
  - Can quickly convert from HEX to binary

Binary	Hex	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Binary Representation

- This representation is based on powers of 2. Any number can be expressed as a string of 0s and 1s

Example: $5 = 101_2 = 1^* 2^2 + 0^*2^1 + 1^*2^0$

Example: $9 = 1001_2 = 1^* 2^3 + 0^* 2^2 + 0^*2^1 + 1^*2^0$

Exercise: Convert the numbers
19, 38, 58 from decimal to binary.
(use an envelope, a calculator or computer program)

Hexadecimal Representation

- This representation is based on powers of 16. Any number can be expressed in terms of:

0,1,2,…,9,A,B,C,D,E,F (0,1,2,…,9,10,11,12,13,14,15)

Example: $256 = 100_{16} = 1 * 16^2 + 0 * 16^1 + 0 * 16^0$

Example: $1002 = 3EA_{16} = 3 * 16^2 + 14 * 16^1 + 10 * 16^0$

Exercise: Convert the numbers
`1492, 3481, 558` from decimal to hex.
(use calculator or maths package)

HEX/Binary Conversion

- Converting HEX/Binary to Decimal is a bit painful, but converting HEX to Binary is trivial
- We often use the prefix `0x` or the suffix h to represent a HEX number
  - Example `0x15AB` for the HEX number 15AB
  - `5Ch` is the HEX number `5c` (or `0x5c`)

Exercise: Convert the numbers
`0xDEAD` `0xBEEF` from Hex to binary.
Now convert them to Decimal

8 Bit Microprocessors

- The PIC microprocessor we will be using is an “8-bit” processor
- The operations the processor performs work on 8-bit numbers
  - Data is copied around the microprocessor internally 8 bits at a time
  - Operations (addition/subtraction/etc..) can be performed on the 8-bit numbers
  - We can of course do calculation with bigger numbers, but we will have to do this as a sequence of operations on 8-bit numbers
    - We will see how to do this later – using a “carry” bit

8 Bit representations

- 8 bits can be used to represent any set of 256 different symbols
- The numbers 0-255
- The numbers -128 to + 127
- Part of a longer number
- A “Character” in ASCII code
  - Hence “char” in C

Unicode uses more bits to code non-Roman characters eg UTF-16

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.asciitable.com

Bytewise and Bitwise Operations

- The processor can perform several operations on 8-bit data
  - Including "Boolean" algebra

Operation	Register f	Register W	Result
addwf	00001111	00000001	00010000
comf	01010101		10101010
iorwf	00010101	00101010	00111111
rlncf	00001111		00011110
rrncf	00001111		00000111
andwf	01010101	10101010	00000000

Exercise: (1) Find NOT(0xAAA)
(2) Find OR(0xAAA; 0x5
(3) Find AND(0xAEB123;

Why is shift (rotate) important ?
Try SHIFTL(011) SHIFTR(011)
(what do SHIFTR and SHIFTL do in base 10?)

What About Negative Numbers?

- With 4 bits, you can represent
  - 0 to +15
  - -8 to + 7
    - There are three ways to represent these negative numbers

- Sign/Magnitude: Set the top bit to 1
- 1's complement : Take the complement of the number
- 2's complement : Take the complement of the number and then add 1

Integer	Sign Magnitude	1's complement	2's complement
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
0	0000	0000	0000
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	1000 (-0)	0111(??)	1000

Subtraction = Addition with Negative Numbers

- There is a good reason to use a 2's complement representation
  - Binary addition of two's complement numbers "just works" whether the numbers are positive or negative

3 + 4	3 + -4	-3 + 4	-3 + -4
0011	0011	1101	1101
+ 0100	+ 1100	+ 0100	+ 1100
= 0111	= 1111	= 0001	= 1001
(7)	(-1)	(+1)	(-7)

Think "Number Lines" 2's complement numbers sit on a number line

Exercise: Fill out the same table using sign magnitude numbers. Do you understand now why 2's complement is a useful representation!

Back to our program

16-bit instructions

```
0100   0E00      MOVLW 0x0
0102   D003      BRA 0x10A
0104   C006      MOVFF 0x06, PORTC
0106   FF82      NOP
0108   2806      INCF 0x06, W, ACCESS
010A   6E06      MOVWF 0x06, ACCESS
010C   0E63      MOVLW 0x63
010E   6406      CPFSGT 0x06, ACCESS
0110   D7F9      BRA 0x104
0112   EF00      GOTO 0x0
0114   F000      NOP
```

- The program is a list of 16-bit instructions stored in *program memory*
- There are 128 Kilobytes of program memory (2^17 bytes)
- Each location stores an 8 bit number
- The location of each *Byte* is specified with a 17 bit *Byte Address*
  - (0x0000-0x1FFFF)

Byte Address	Instruction Address	Value
0100	0080	00
0101		0E
0102	0081	03
0103		D0

Little-endian: LSB stored first

Registers on the PIC

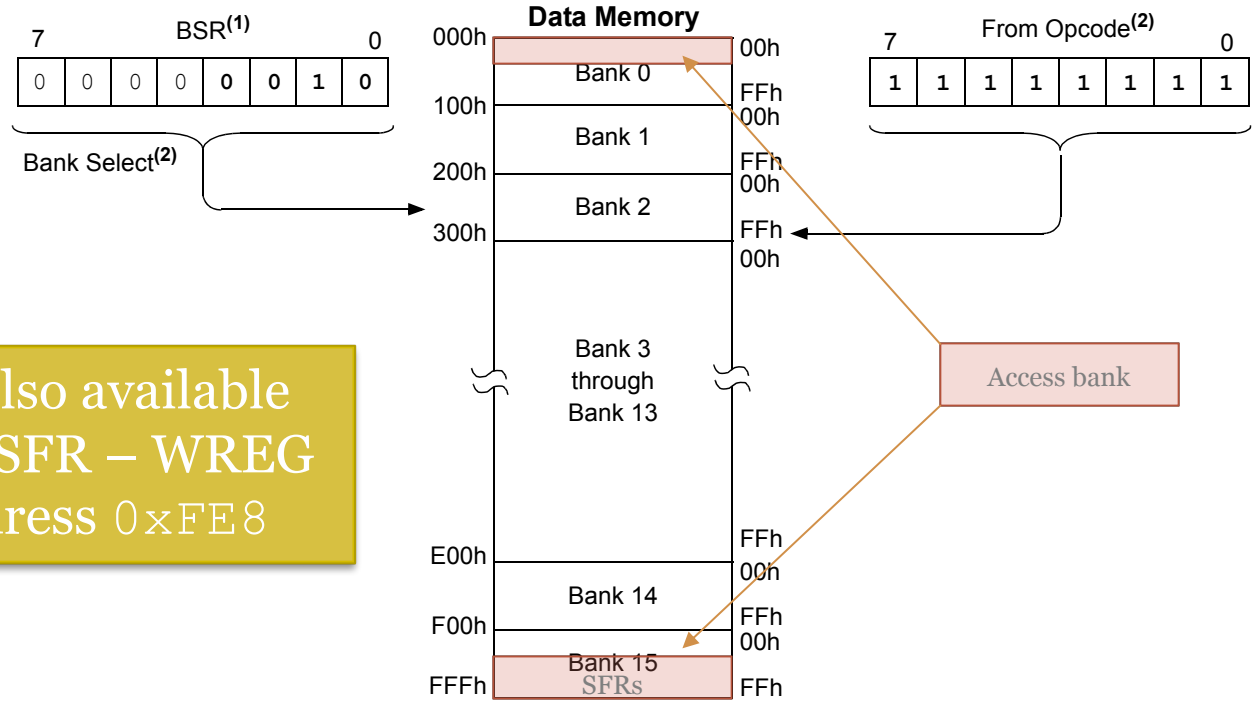
The PIC has one special "working register" W that can be used as an operand and/or destination in many instructions

The whole of on-chip RAM (4k) can be used as the other operand and/or destination (f) in many instructions. These are known as "file registers"

Access to RAM locations is via a 12-bit address – But many instructions only use a 8-bit address to access ram within one of 16 BANKs.

The 4 bits for the selected BANK is stored in the bank select register BSR

- PIC file Registers – where data is stored and numerical work is done in RAM



W is also available as an SFR – WREG at address `0xFE8`

- The "Access Bank" is a special bank comprising the first 96 bytes of Bank 0 (`00h-5Fh`), general storage, and the last 160 bytes of Bank 15 (`60h-FFh`), containing many special function registers (SFRs) that control operation of the microprocessor and its peripherals

Back to our program: Loading a register


```
0100   0E00      MOVLW 0x0
0102   D003      BRA 0x10A
0104   C006      MOVFF 0x06, PORTB
0106   FF81      NOP
0108   2806      INCF 0x06, W, ACCESS
010A   6E06      MOVWF 0x06, ACCESS
010C   0E63      MOVLW 0x63
010E   6406      CPFSGT 0x06, ACCESS
0110   D7F9      BRA 0x104
0112   EF00      GOTO 0x0
0114   F000      NOP
```

- The first instruction is loading a value of `0x00` into W
- This instruction is encoded in the 16-bit opcode stored at address `0x0100`
- The Assembler code is
  - `movlw 0x0`
  - Move the literal value `0x0` into the W register

Decoding an Opcode

MOVLW	**Move Literal to W**
Syntax:	MOVLW k
Operands:	0 ≤ k ≤ 255
Operation:	k → W
Status Affected:	None
Encoding:	0000 | 1110 | kkkk | kkkk
Description:	The eight-bit literal 'k' is loaded into W.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: `MOVLW 5Ah`

After Instruction
W = 5Ah

Words are stored "little endian"
Read into the processor and stored in program memory as `0x00,0x0E`
`0x0E00=0000 1110 0000 0000`
kkkk kkkk=`0x00`

Most instructions run in 4 clock cycles, here:
Q1: Decode instruction
Q2: Read literal 'k'
Q3: Process data
Q4: Write to W

Exercise: what is the opcode for
`movlw 0x3F`

Something more complex…

INCF	**Increment f**
Syntax:	INCF f {,d {,a}}
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(f) + 1 → dest
Status Affected:	C, DC, N, OV, Z
Encoding:	0010 | 10da | ffff | ffff
Description:	The contents of register 'f' are incremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f'. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the GPR bank.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

This instruction adds 1 to the value stored in file register at binary address `ffff ffff`

The file register can come from the Access Bank by setting bit a=0 or from normal banked ram by setting bit a=1

The destination of the result can be chosen as back to the file register with bit d=1 or to the W register with bit d=0

Exercise: what is the opcode for
`incf 0x16,f,ACCESS`

Back to our program

```
0100   0E00      MOVLW 0x0
0102   D003      BRA 0x10A
0104   C006      MOVFF 0x06, PORTB
0106   FF81      NOP
0108   2806      INCF 0x06, W, ACCESS
010A   6E06      MOVWF 0x06, ACCESS
010C   0E63      MOVLW 0x63
010E   6406      CPFSGT 0x06, ACCESS
0110   D7F9      BRA 0x104
0112   EF00      GOTO 0x0
0114   F000      NOP
```

The second instruction jumps (branches 3 instructions ahead) to the instruction at address `0x10A`

Here we move the current value of W into file register `0x06`

Puts a new value 0x63=99 into W

This instruction is comparing the value `in` register `0x06` to W and skips the next instruction if it is greater

If we don't skip then branch back to `0x104`

Output the value in `0x06` to (SFR) PORTB

The instruction `movff` is a 2-word instruction as it has 2 full 12-bit addresses for file registers. So word at `0x106` displays as `NOP` (no operation) although it is the part of `movff` containing the second address

Compare assembly language to c


```
main()
{
    char i;
    i=0;
    while (i<100) {
        PORTB = i;
        i=i+1;
    }
}
```

```
0100  0E00     MOVLW 0x0
0102  D003     BRA 0x10A
0104  C006     MOVFF 0x06, PORTB
0106  FF81     NOP
0108  2806     INCF 0x06, W, ACCESS
010A  6E06     MOVWF 0x06, ACCESS
010C  0E63     MOVLW 0x63
010E  6406     CPFSGT 0x06, ACCESS
0110  D7F9     BRA 0x104
0112  EF00     GOTO 0x0
0114  F000     NOP
```

- Here the variable `i` is stored in file register `0x06`
- Pretty easy to see how this program is translated
- More complex programs quickly become very complicated to understand in assembly language unless you are careful how you structure and comment your code

Programming in this course

- We will be programming exclusively in assembler
  - Allows us to understand precisely what the microprocessor is going to do and how long it will take to do so
    - important for time critical applications
  - Full access to all functions of the microprocessor
  - With care can make very efficient use of resources
    - Sometimes very important for small microprocessors
- Programming in assembler requires some discipline
  - Code can be very difficult to understand
    - The code is *very* low level
    - Line by line comments are very important

The PIC instruction set

- We've seen a few sample instructions which cover most of the basic type of operations
- Arithmetic and Logic instructions
  - (ADD, SUB, AND, OR, EOR, COM, INC, DEC, …)
- Branch Instructions
  - Jump to a different location depending on a test
- Data transfer instructions
  - Move data around the microprocessor
- Bit setting and testing operations
  - Manipulate and test bits in registers

Data Transfer Instructions

- Move

```
movlw 27
W ← 27
movwf 0x06,A
Access(0x06) ← W
```

- Input

```
movf PORTE,W
W ← PORTE
```

- Bank selection

```
movlb 3
BSR ← 3
```

- Copy file register

```
movff 0x178,0x29
(0x29) ← (0x178)
```

- Output

```
movwf PORTD
PORTD ← W
```

All file registers and special function registers (SFR) look the same, even W appears as WREG in RAM

Arithmetic and Logic Instructions

- Addition

```
addwf 0x08,W
W ← (0x08)+W
```

- Subtraction

```
subwf 0x54,f
(0x54) ← (0x54)-W
```

- Increment

```
incf 0x17,f
(0x17) ← (0x17)+1
```

- Logic

```
andwf 0x03,W
W ← AND((0x03),W)
```

Remember that the file register can be either from a defined bank or the Access bank

The Status Register

REGISTER 6-2: STATUS REGISTER

U-0	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	N	OV	Z	DC(1)	C(2)
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-5 **Unimplemented:** Read as '0'

bit 4 **N:** Negative bit
This bit is used for signed arithmetic (2's complement). It indicates whether the result was negative (ALU MSB = 1).
1 = Result was negative
0 = Result was positive

bit 3 **OV:** Overflow bit
This bit is used for signed arithmetic (2's complement). It indicates an overflow of the seven-bit magnitude which causes the sign bit (bit 7) to change state.
1 = Overflow occurred for signed arithmetic (in this arithmetic operation)
0 = No overflow occurred

bit 2 **Z:** Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC:** Digit Carry/$\overline{\text{Borrow}}$ bit(1)
For ADDWF, ADDLW, SUBLW and SUBWF instructions:
1 = A carry-out from the 4th low-order bit of the result occurred
0 = No carry-out from the 4th low-order bit of the result

bit 0 **C:** Carry/$\overline{\text{Borrow}}$ bit(2)
For ADDWF, ADDLW, SUBLW and SUBWF instructions:
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

- For many operations the microprocessor sets bits in the Status Register (STATUS)
  - `addlw 0x77`
- STATUS is an SFR and can be examined/manipulated as any other file register - but be careful
  - `addwf STATUS,f`
- The STATUS bits can be tested by branch instructions to decide whether or not to jump to a different location

Branch/Control Instructions

- Branch

  `bnz label2`

  Branch if not zero to location `label2`

  If the Branch test fails – the next line of code is executed

  If the Branch test is successful, the program jumps to the location specified

  Branch operations are relative jumps and can only jump by -128 to +127 instructions from the position of the next instruction.

- Goto

  `goto label3`

  Jump to label3

  This instruction can jump to anywhere in program memory but is a 2-word instruction

- Call, Return

  `rcall mysub`

  `return`

  Call subroutine mysub. The program saves information about where it is currently executing and then jumps to the code at mysub. When the subroutine is finished it executes `return`, which then returns to where the `rcall` was made.

Arithmetic and Logic Instructions

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word MSb LSb	Status Affected
BYTE-ORIENTED OPERATIONS					
ADDWF	f, d, a	Add WREG and f	1	0010 01da ffff ffff	C, DC, Z, OV, N
ADDWFC	f, d, a	Add WREG and Carry bit to f	1	0010 00da ffff ffff	C, DC, Z, OV, N
ANDWF	f, d, a	AND WREG with f	1	0001 01da ffff ffff	Z, N
CLRF	f, a	Clear f	1	0110 101a ffff ffff	Z
COMF	f, d, a	Complement f	1	0001 11da ffff ffff	Z, N
CPFSEQ	f, a	Compare f with WREG, Skip =	1 (2 or 3)	0110 001a ffff ffff	None
CPFSGT	f, a	Compare f with WREG, Skip >	1 (2 or 3)	0110 010a ffff ffff	None
CPFSLT	f, a	Compare f with WREG, Skip <	1 (2 or 3)	0110 000a ffff ffff	None
DECF	f, d, a	Decrement f	1	0000 01da ffff ffff	C, DC, Z, OV, N
DECFSZ	f, d, a	Decrement f, Skip if 0	1 (2 or 3)	0010 11da ffff ffff	None
DCFSNZ	f, d, a	Decrement f, Skip if Not 0	1 (2 or 3)	0100 11da ffff ffff	None
INCF	f, d, a	Increment f	1	0010 10da ffff ffff	C, DC, Z, OV, N
INCFSZ	f, d, a	Increment f, Skip if 0	1 (2 or 3)	0011 11da ffff ffff	None
INFSNZ	f, d, a	Increment f, Skip if Not 0	1 (2 or 3)	0100 10da ffff ffff	None
IORWF	f, d, a	Inclusive OR WREG with f	1	0001 00da ffff ffff	Z, N
MOVF	f, d, a	Move f	1	0101 00da ffff ffff	Z, N
MOVFF	f_s, f_d	Move f_s (source) to 1st word f_d (destination) 2nd word	2	1100 ffff ffff ffff 1111 ffff ffff ffff	None
MOVWF	f, a	Move WREG to f	1	0110 111a ffff ffff	None
MULWF	f, a	Multiply WREG with f	1	0000 001a ffff ffff	None
NEGF	f, a	Negate f	1	0110 110a ffff ffff	C, DC, Z, OV, N
RLCF	f, d, a	Rotate Left f through Carry	1	0011 01da ffff ffff	C, Z, N
RLNCF	f, d, a	Rotate Left f (No Carry)	1	0100 01da ffff ffff	Z, N
RRCF	f, d, a	Rotate Right f through Carry	1	0011 00da ffff ffff	C, Z, N
RRNCF	f, d, a	Rotate Right f (No Carry)	1	0100 00da ffff ffff	Z, N
SETF	f, a	Set f	1	0110 100a ffff ffff	None
SUBFWB	f, d, a	Subtract f from WREG with Borrow	1	0101 01da ffff ffff	C, DC, Z, OV, N
SUBWF	f, d, a	Subtract WREG from f	1	0101 11da ffff ffff	C, DC, Z, OV, N
SUBWFB	f, d, a	Subtract WREG from f with Borrow	1	0101 10da ffff ffff	C, DC, Z, OV, N
SWAPF	f, d, a	Swap Nibbles in f	1	0011 10da ffff ffff	None
TSTFSZ	f, a	Test f, Skip if 0	1 (2 or 3)	0110 011a ffff ffff	None
XORWF	f, d, a	Exclusive OR WREG with f	1	0001 10da ffff ffff	Z, N

Bit oriented and control Instructions

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word MSb LSb	Status Affected
BIT-ORIENTED OPERATIONS					
BCF	f, b, a	Bit Clear f	1	1001 bbba ffff ffff	None
BSF	f, b, a	Bit Set f	1	1000 bbba ffff ffff	None
BTFSC	f, b, a	Bit Test f, Skip if Clear	1 (2 or 3)	1011 bbba ffff ffff	None
BTFSS	f, b, a	Bit Test f, Skip if Set	1 (2 or 3)	1010 bbba ffff ffff	None
BTG	f, b, a	Bit Toggle f	1	0111 bbba ffff ffff	None
CONTROL OPERATIONS					
BC	n	Branch if Carry	1 (2)	1110 0010 nnnn nnnn	None
BN	n	Branch if Negative	1 (2)	1110 0110 nnnn nnnn	None
BNC	n	Branch if Not Carry	1 (2)	1110 0011 nnnn nnnn	None
BNN	n	Branch if Not Negative	1 (2)	1110 0111 nnnn nnnn	None
BNOV	n	Branch if Not Overflow	1 (2)	1110 0101 nnnn nnnn	None
BNZ	n	Branch if Not Zero	1 (2)	1110 0001 nnnn nnnn	None
BOV	n	Branch if Overflow	1 (2)	1110 0100 nnnn nnnn	None
BRA	n	Branch Unconditionally	2	1101 0nnn nnnn nnnn	None
BZ	n	Branch if Zero	1 (2)	1110 0000 nnnn nnnn	None
CALL	n, s	Call Subroutine 1st word	2	1110 110s kkkk kkkk	None
		2nd word		1111 kkkk kkkk kkkk	
CLRWDT	—	Clear Watchdog Timer	1	0000 0000 0000 0100	$\overline{TO}$, $\overline{PD}$
DAW	—	Decimal Adjust WREG	1	0000 0000 0000 0111	C
GOTO	n	Go to Address 1st word	2	1110 1111 kkkk kkkk	None
		2nd word		1111 kkkk kkkk kkkk	
NOP	—	No Operation	1	0000 0000 0000 0000	None
NOP	—	No Operation	1	1111 xxxx xxxx xxxx	None
POP	—	Pop Top of Return Stack (TOS)	1	0000 0000 0000 0110	None
PUSH	—	Push Top of Return Stack (TOS)	1	0000 0000 0000 0101	None
RCALL	n	Relative Call	2	1101 1nnn nnnn nnnn	None
RESET		Software Device Reset	1	0000 0000 1111 1111	All
RETFIE	s	Return from Interrupt Enable	2	0000 0000 0001 000s	GIE/GIEH, PEIE/GIEL
RETLW	k	Return with Literal in WREG	2	0000 1100 kkkk kkkk	None
RETURN	s	Return from Subroutine	2	0000 0000 0001 001s	None
SLEEP	—	Go into Standby mode	1	0000 0000 0000 0011	$\overline{TO}$, $\overline{PD}$

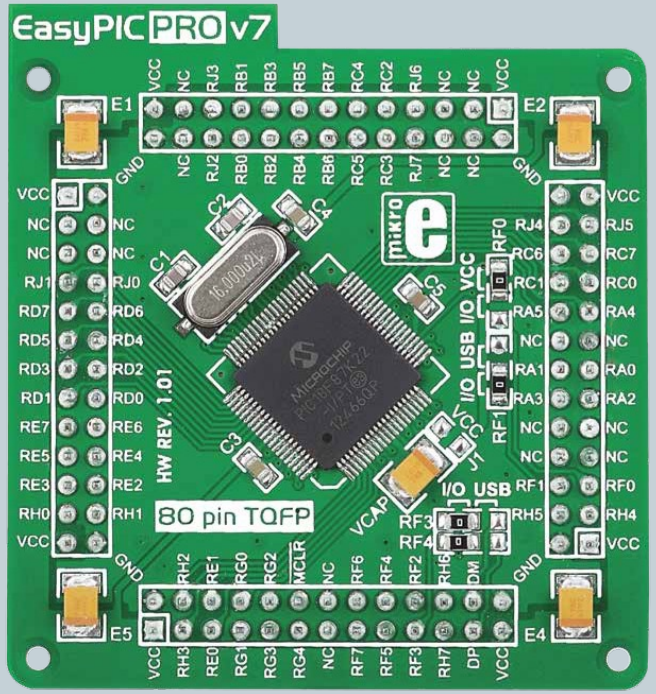
Literal and data transfer instructions

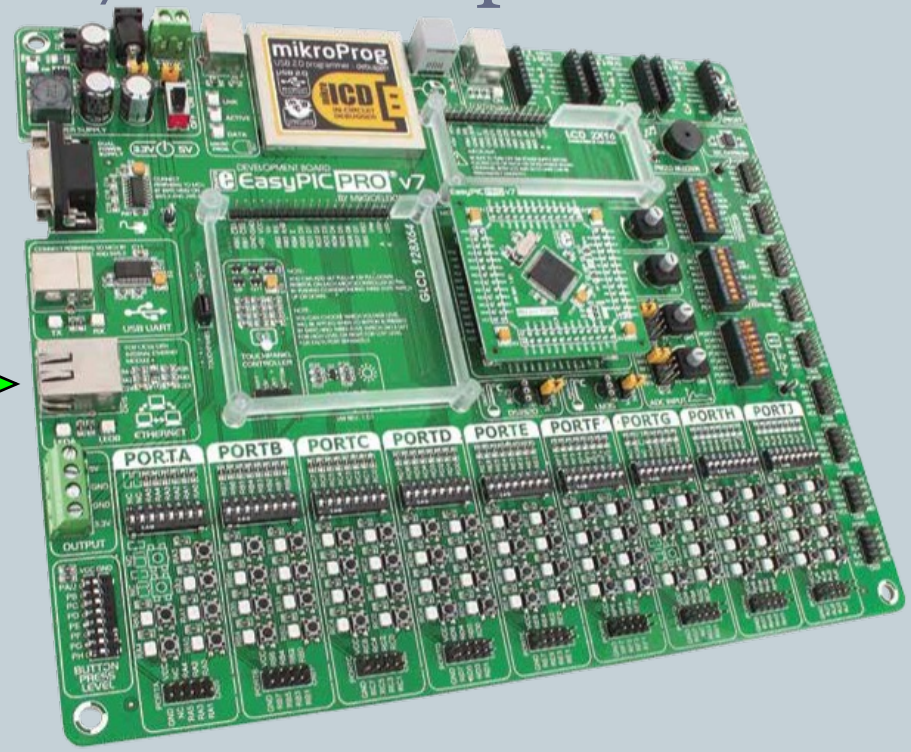
Mnemonic, Operands	Description	Cycles	16-Bit Instruction Word MSb LSb	Status Affected	Notes
LITERAL OPERATIONS					
ADDLW k	Add Literal and WREG	1	0000 1111 kkkk kkkk	C, DC, Z, OV, N	
ANDLW k	AND Literal with WREG	1	0000 1011 kkkk kkkk	Z, N	
IORLW k	Inclusive OR Literal with WREG	1	0000 1001 kkkk kkkk	Z, N	
LFSR f, k	Move literal (12-bit) 2nd word to FSR(f) 1st word	2	1110 1110 00ff kkkk 1111 0000 kkkk kkkk	None	
MOVLB k	Move Literal to BSR<3:0>	1	0000 0001 0000 kkkk	None	
MOVLW k	Move Literal to WREG	1	0000 1110 kkkk kkkk	None	
MULLW k	Multiply Literal with WREG	1	0000 1101 kkkk kkkk	None	
RETLW k	Return with Literal in WREG	2	0000 1100 kkkk kkkk	None	
SUBLW k	Subtract WREG from Literal	1	0000 1000 kkkk kkkk	C, DC, Z, OV, N	
XORLW k	Exclusive OR Literal with WREG	1	0000 1010 kkkk kkkk	Z, N	
DATA MEMORY ↔ PROGRAM MEMORY OPERATIONS					
TBLRD*	Table Read	2	0000 0000 0000 1000	None	
TBLRD*+	Table Read with Post-Increment		0000 0000 0000 1001	None	
TBLRD*-	Table Read with Post-Decrement		0000 0000 0000 1010	None	
TBLRD+*	Table Read with Pre-Increment		0000 0000 0000 1011	None	
TBLWT*	Table Write	2	0000 0000 0000 1100	None	
TBLWT*+	Table Write with Post-Increment		0000 0000 0000 1101	None	
TBLWT*-	Table Write with Post-Decrement		0000 0000 0000 1110	None	
TBLWT+*	Table Write with Pre-Increment		0000 0000 0000 1111	None	

Find all this information (and more) in the PIC18F87K22 datasheet on the microprocessors section of the third year lab website

The PIC18F87K22 Microprocessor

- In this course we will be using a PIC18F87K22 mounted on a Mikroelectronika EasyPIC PRO V7.0 development board





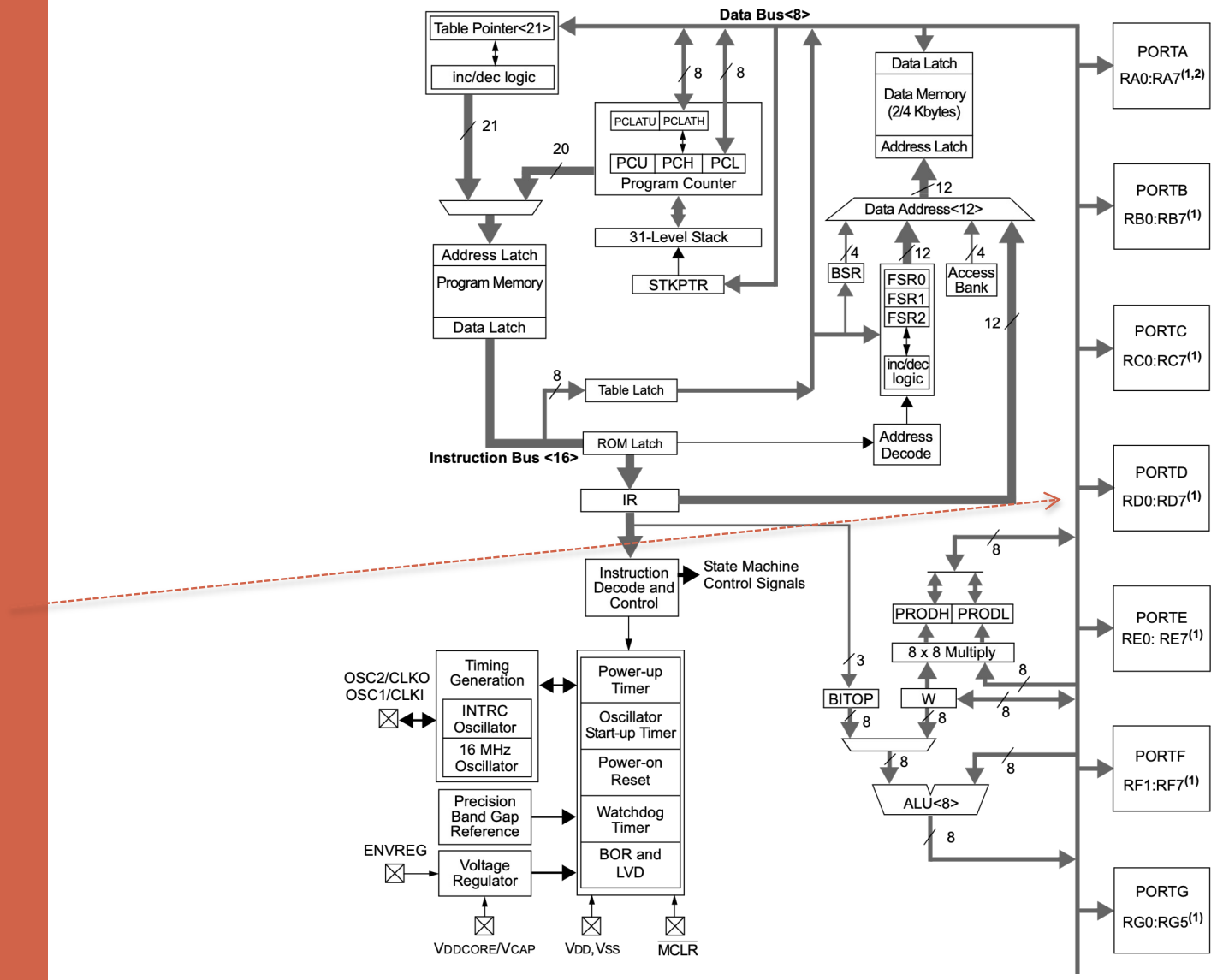
The PIC18F87K22

The microprocessor you will be using has several input and output ports

These are already connected to switches or LEDs on the boards we are using

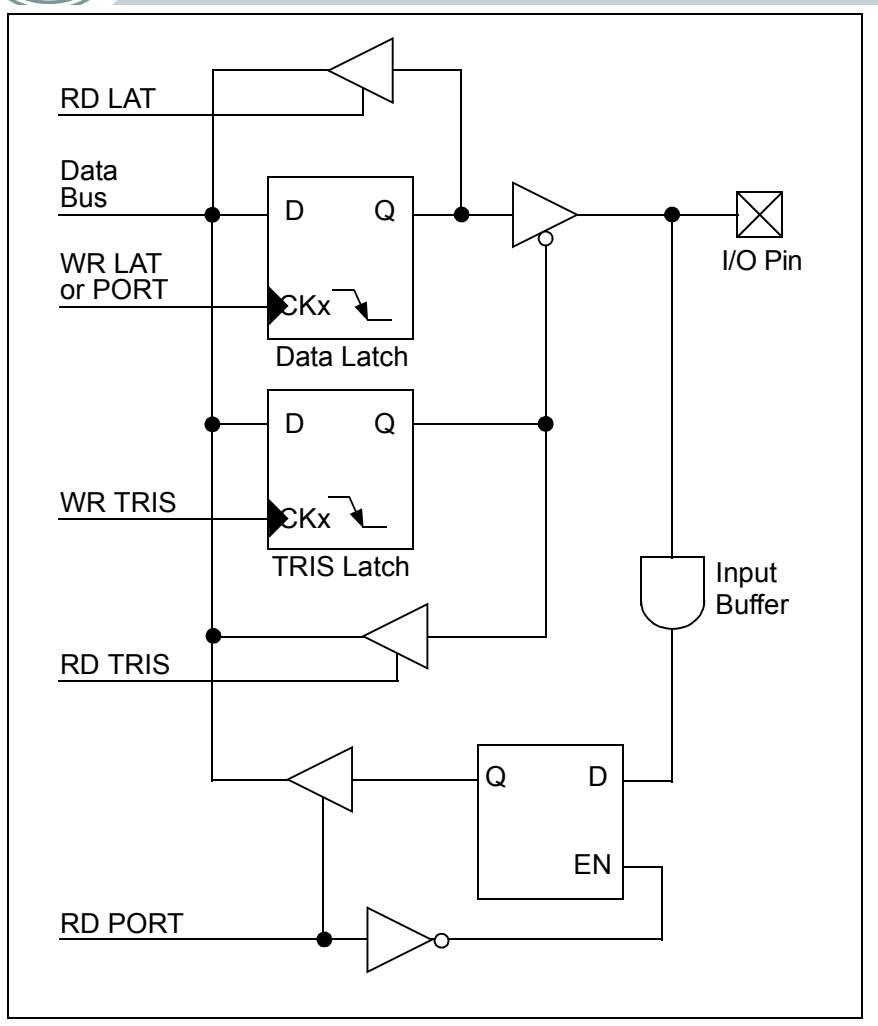
They are also available to you to connect to external devices.

It is time to make some lights blink!



Setting up the Input/Output ports:

- For the ports we set the individual pin to be input or outputs using a port's tri-state register (eg `TRISB`) which has a bit for each bit of I/O (a 1 turns the output off so you can use it as an input)
- Each port also has a port register where we can write to or read the status of the port pins (eg `PORTF`).
- And a latch register where we can write data to be output (eg `LATA`), useful for Read/Modify/Write operations
- NB: Some Port pins can also be *analog inputs* and *default to analog* on reset. You can switch them to digital using the ANCON* special function registers



Setting up the Input/Output ports:


```
; ******* Port B Setup Code ****
movlw    0x00          ; all bits out
movwf    TRISB, A      ; Port B TRIS Register
movlw    0x39          ; Init value
movwf    LATB, A       ; Port B value

; ******* Port D Setup Code ****
movlw    0xff          ; all bits in
movwf    TRISD, A      ; Port D Direction Register
movlb    0x0f          ; PADCFG1 (and RDPU) are not in access RAM
bsf      RDPU          ; Turn on pull-ups for Port D
```

The TRIS register essentially sets pins into a high impedance tri-state mode (neither `0` nor `1`), in this mode other external signals can pull the pin low or high.

Additionally, pins on ports B, D, E and J can have weak pull-up resistors turned on that pull their voltage to high, allowing other signals to pull it low if they want

Predefined symbols in assembler

- Symbols like PORTA, TRISC, LATD are predefined in a file pic18f87k22.inc that is included from xc.inc at the top of the file that you will see in a moment.
- Even macros for single bit operands are predefined for you such as RB3 (bit 3 of PORTB) and RDPU (bit 7 of PADCFG1)
  - Note that PADCFG1 is not in ACCESS ram
- Even BANKMASK is a macro! As is BANKSEL.

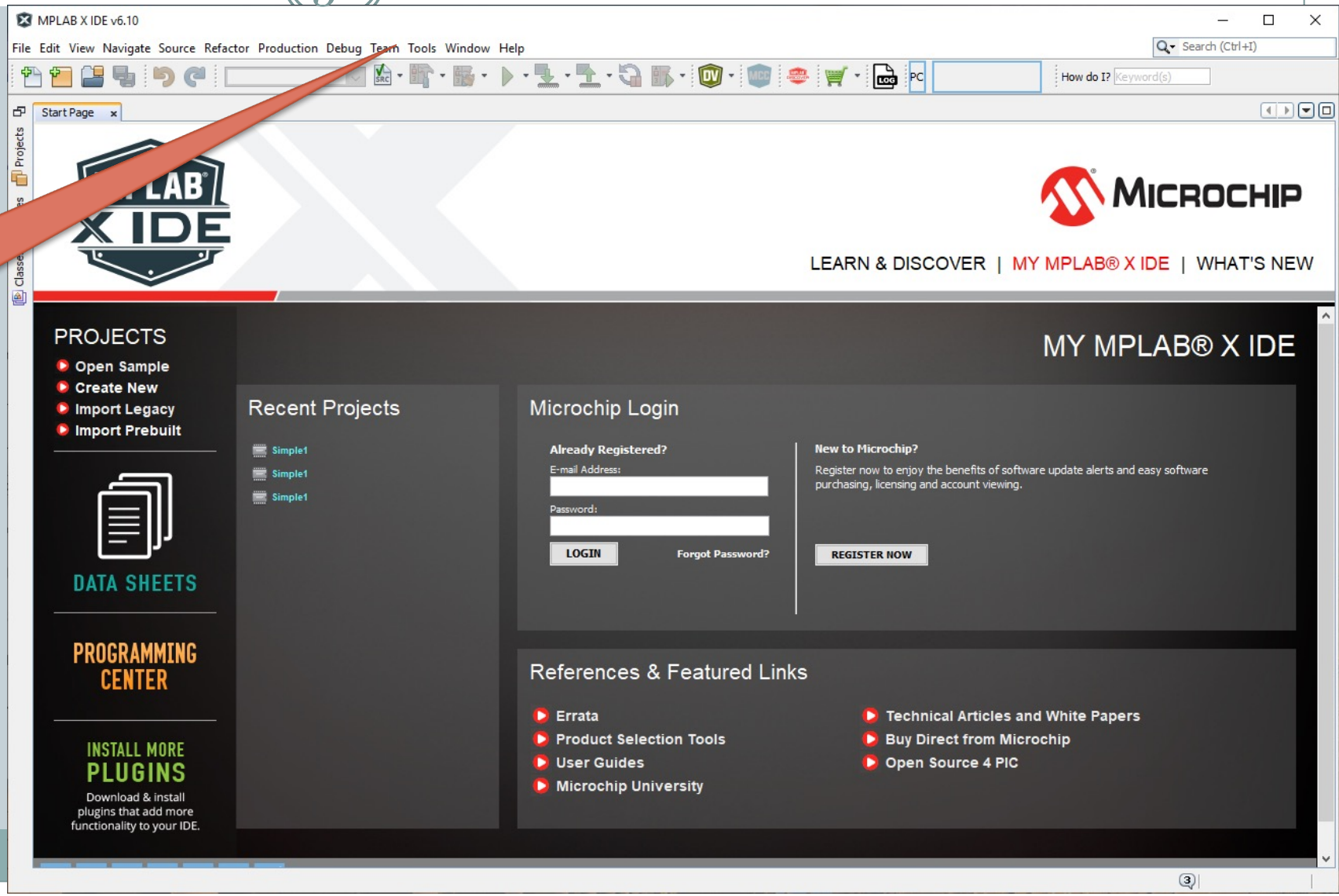
Examples of predefined symbols in pic18f87k22.inc

```
PORTA           equ 0F80h
TRISC           equ 0F94h
LATD            equ 0F8Ch

#define RB3   BANKMASK(PORTB), 3, a
#define RDPU  BANKMASK(PADCFG1), 7, b

#define BANKMASK(addr) ((addr) and 0FFh)
```

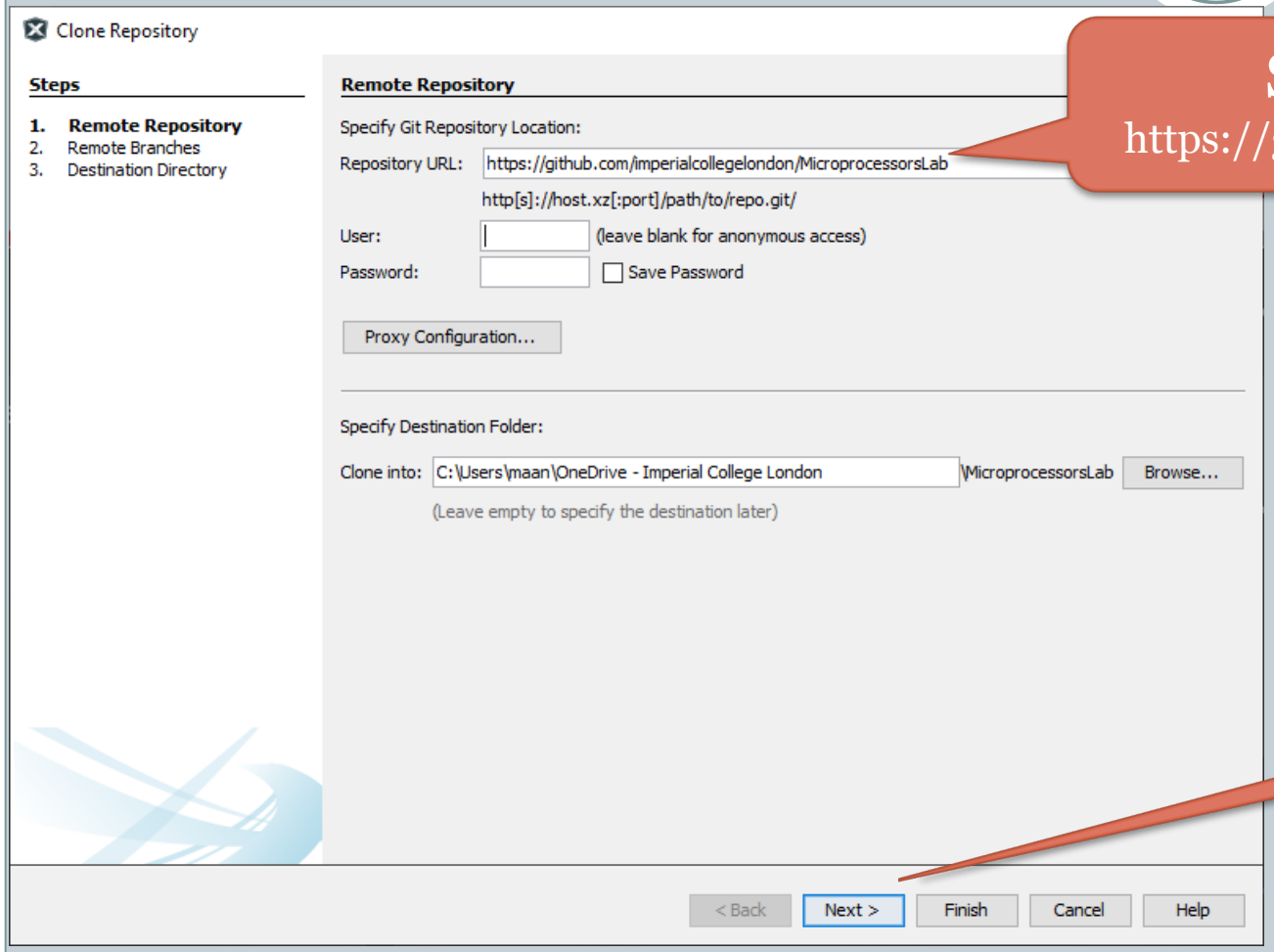
Getting Started with MPLAB-X:

- Start MPLAB X IDE

Under Team select “Git-> clone”



Git clone dialog 1

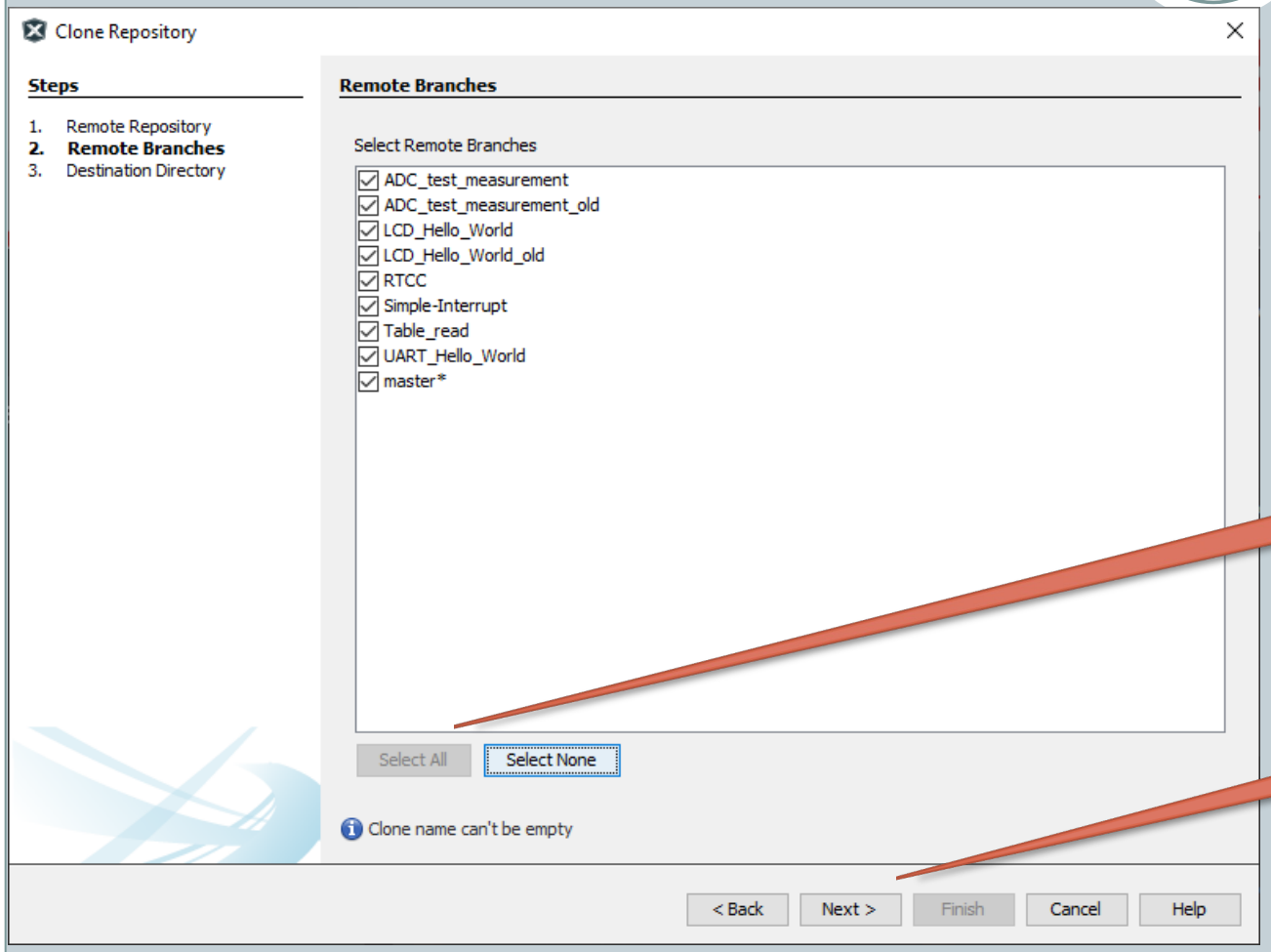


Select the course code repository:
https://github.com/imperialCollegeLondon/MicroprocessorsLab

And a directory on your OneDrive to store your clone

Click Next

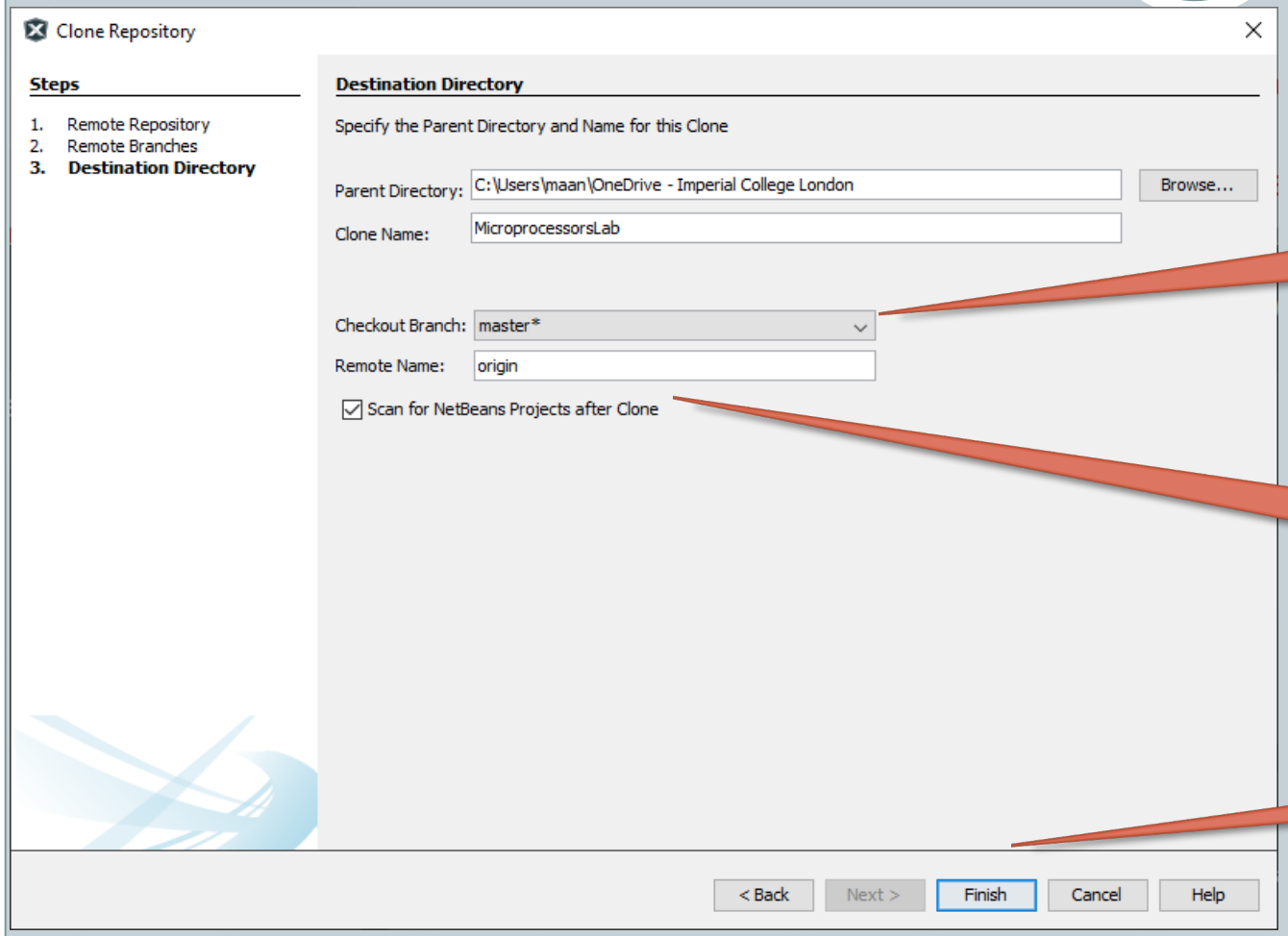
Git clone dialog 2



Choose “Select All”

Click Next

Git clone dialog 3



Choose Checkout Branch master

Select “Scan for projects…”

Click Finish

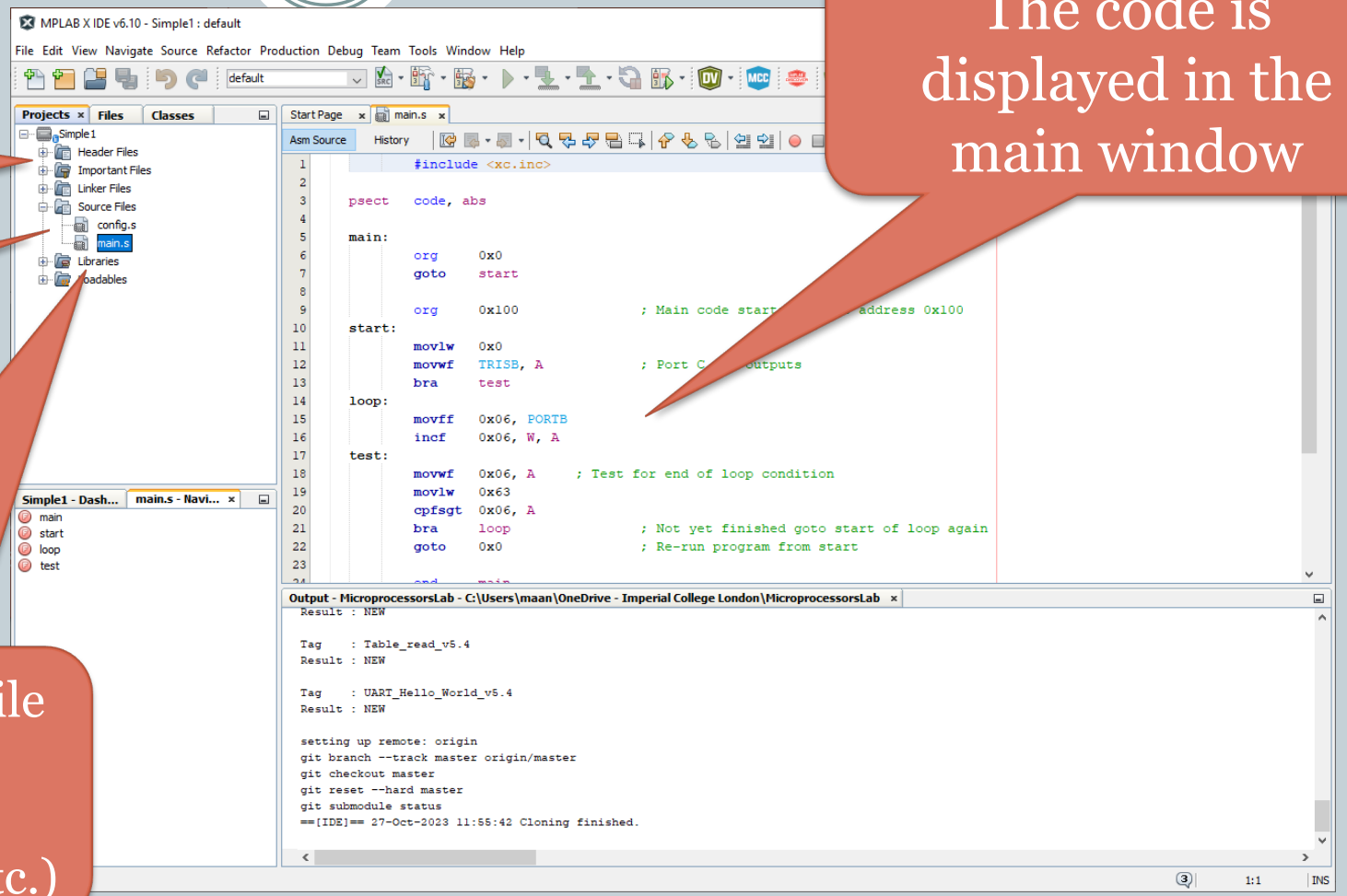
MPLAB up and running

Your starting project is now loaded

Navigate the project structure to find file ”main.s”

The project also contains a file named ”config.s” that configures some of the processor internals (clocks etc.)

The code is displayed in the main window

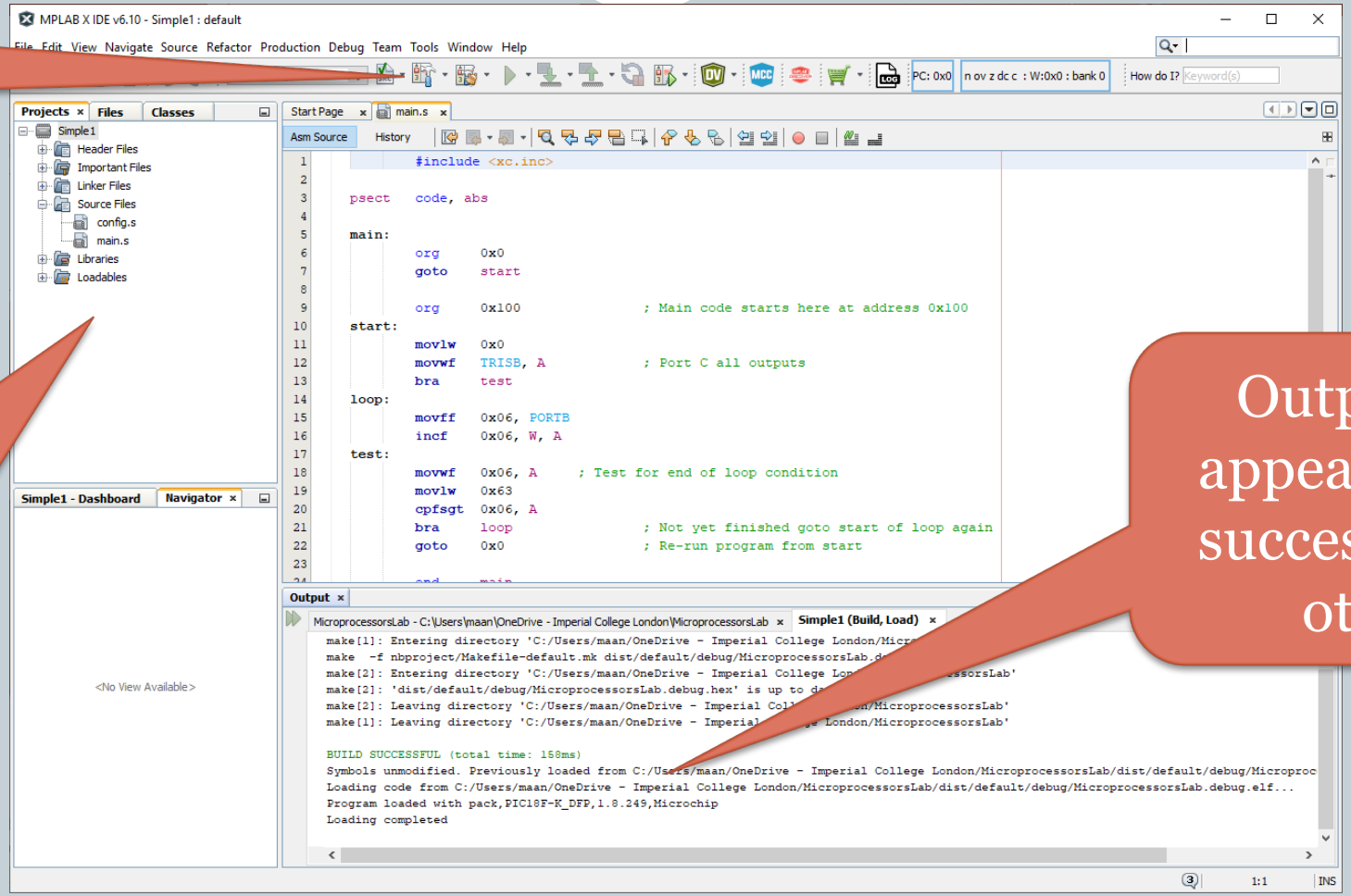


What have you just done?

- Git (and Github) is a source code control system
  - Git is a program you run on your computer to manage changes to your code
  - Github is a cloud storage website where you can store and track complete projects
- Properly used it can be a very useful tool in code development
- It allows you to track changes in your code
  - So that you can go backwards, if you want
- It enables collaboration within teams developing different parts of a larger project
  - Just like you will be doing later in the course
- You have just made a local copy (clone) of a “repository” containing your starting project
- More on Git later…

Building and running your program

Click on Build



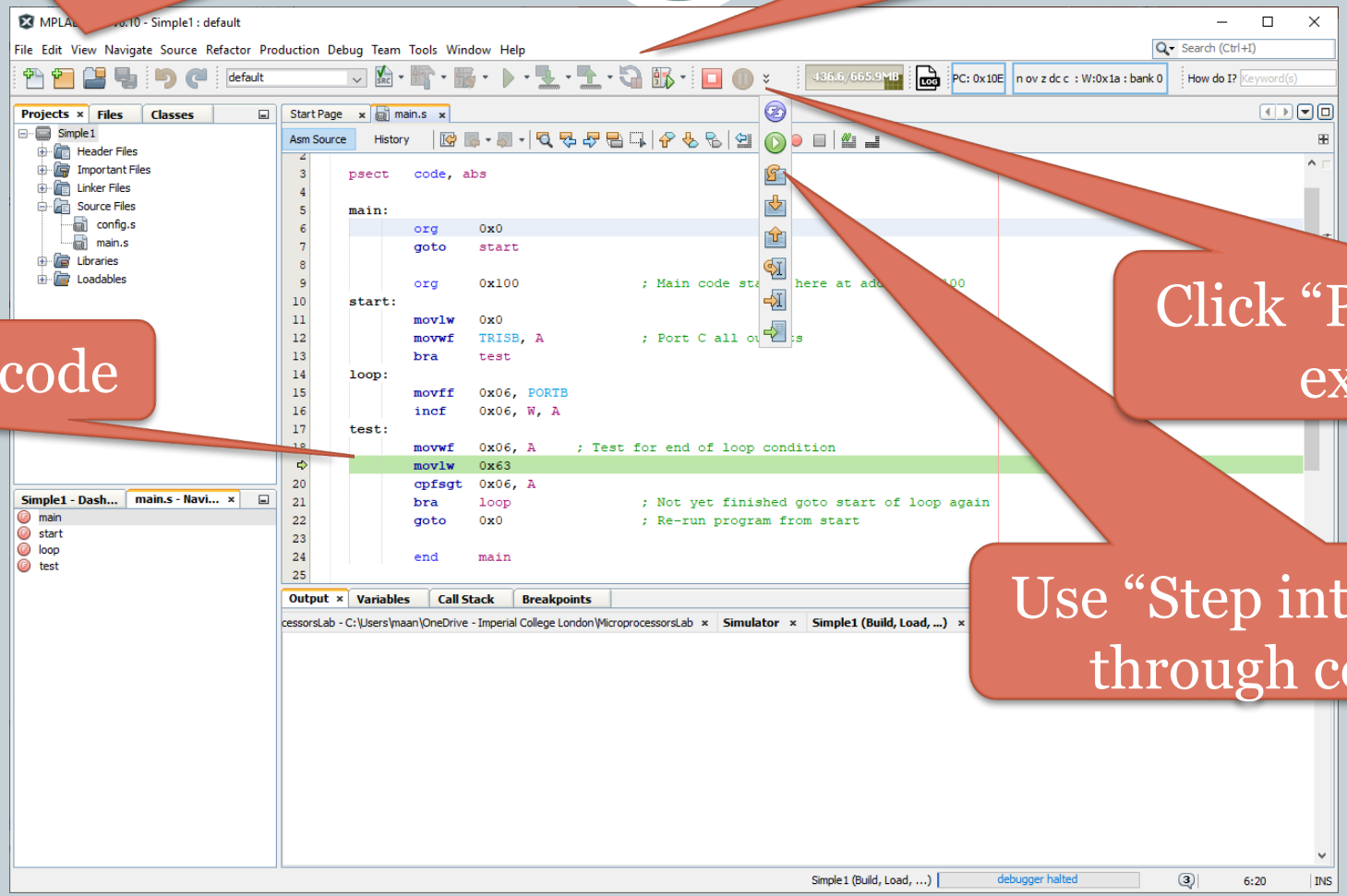
Some more files will appear in your project structure

Output window appears showing a successful build or otherwise!

d running your program

Look in View:Toolbars to show/hide different buttons

Hit “Start debugging” button



Current point in code

Click “Pause” to stop execution

Use “Step into” button to step through code, or key F7

Debug views

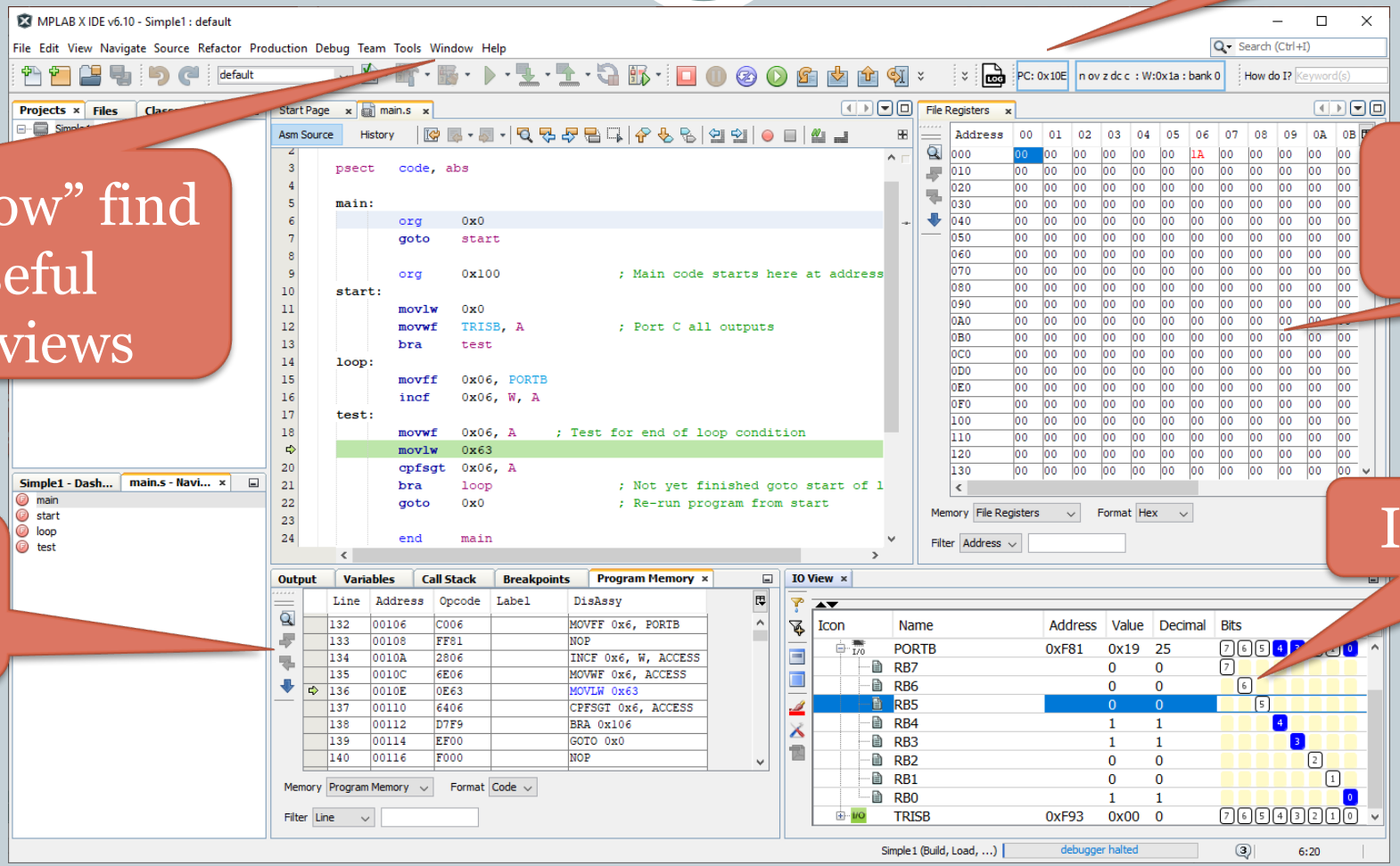
Processor status

Under ”Window” find several useful debugging views

Data memory (file registers)

Program memory

IO port views



Exercising the PIC18 commands:

- Choose “Debug project” from the Debug menu then “Pause” the programme
- Step through the programme and make sure you understand what is happening after each instruction
  - F7 key or the “Step into” menu item
- Try changing the number of times the loop is executed and make sure you understand the outputs on `PORTB`

Source code control…

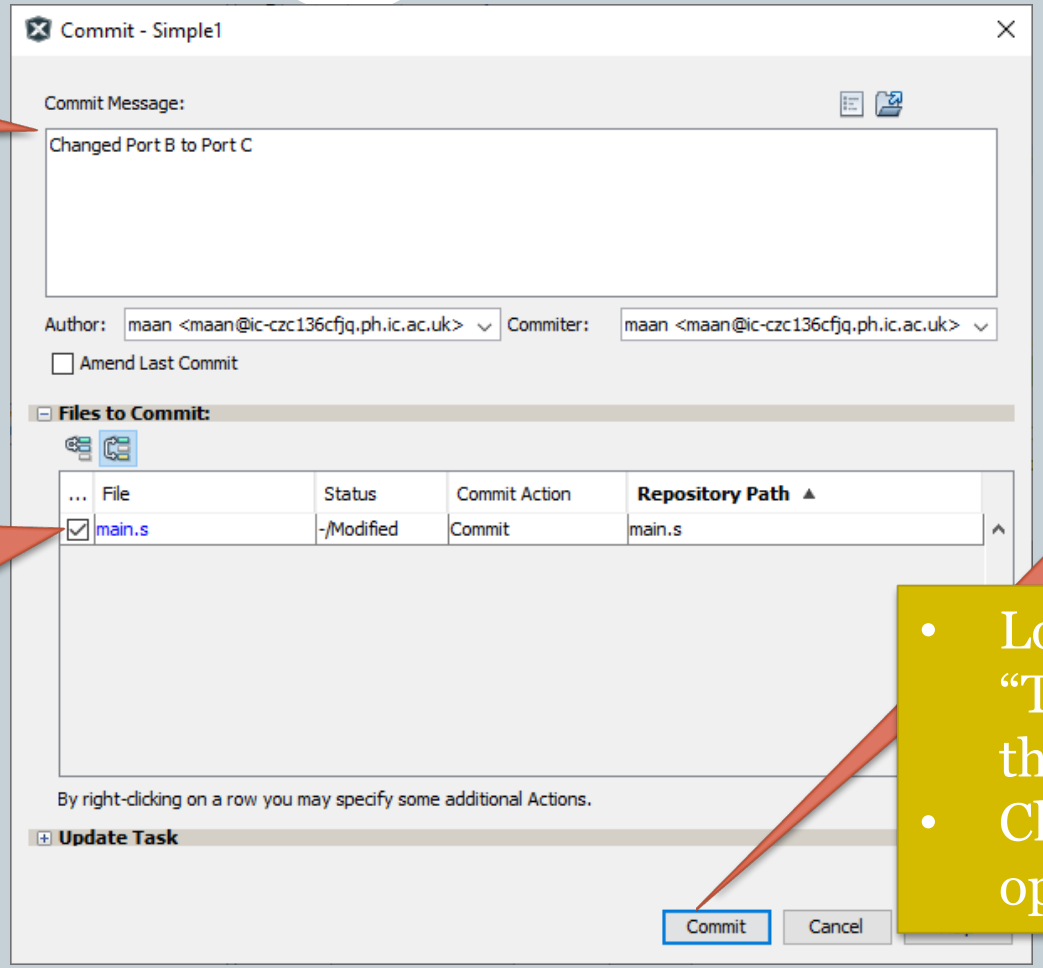
- Modify your code to use `PORTC` instead of `PORTB` Check that it all runs as expected

- You now want to save those changes permanently to your local repository
- Right click on your project in the side bar and choose “Commit…” under “Git” in the menu

Git commit…

Add a useful message to say what changes have been made

Files with changes are listed here, choose the ones you want to include in this commit



Hit commit

- Look through the "Team" menu at other things you can do
- Check out the "history" option

Connecting up the development board

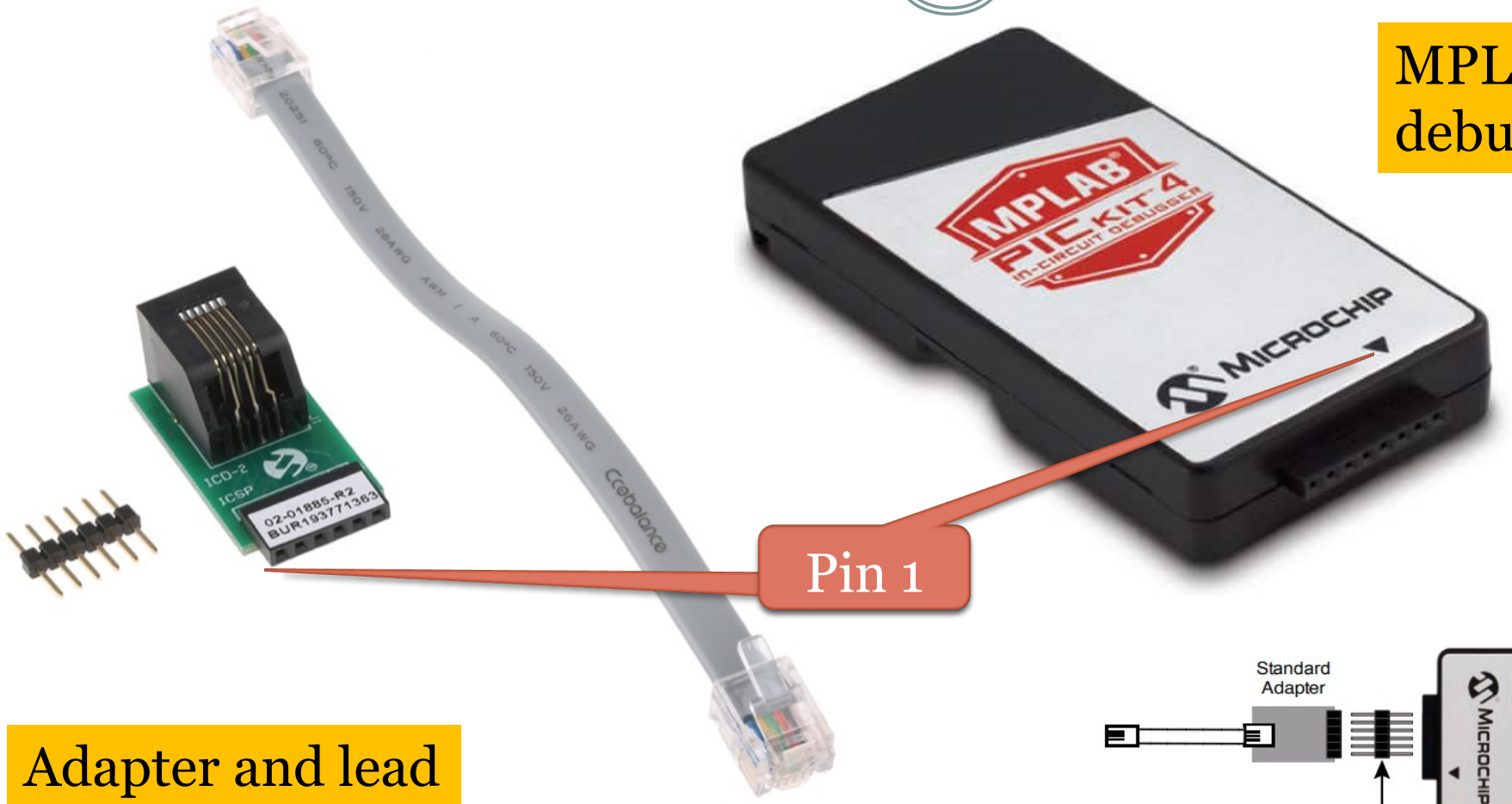
Power connector and switch

ICD interface: Programming and debugging

Note we don't use the on board ICD, but an external microchip ICD instead connected to computer by USB

Alternative PICKIT4 ICD connections

MPLAB PicKit4 debugger

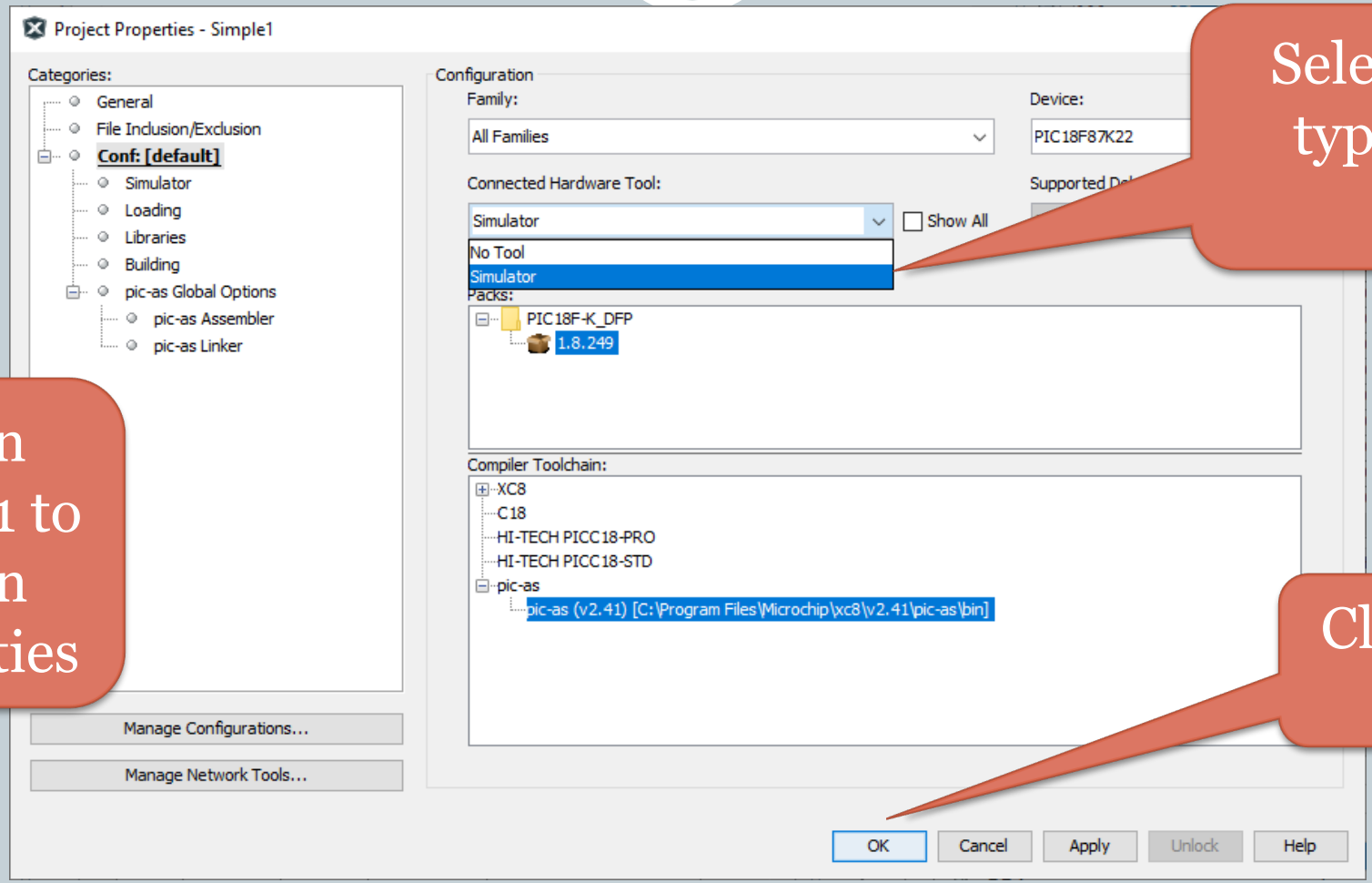


Adapter and lead

What does the In Circuit Debugger do?

- The ICD connector provides an interface to your microprocessor that lets you
  - Download from main computer to microprocessor device memory (program and data)
  - Control execution of code in microprocessor (start/stop/step)
  - Read device memory and register contents back to computer
- ICD is a useful tool not just for its ability to program your microprocessor but also to let you see how your program is working on the device in its particular hardware environment
  - Essentially it can do what you did with the simulator but live in your circuit
- The ICD uses pins `RB6` and `RB7` from `PORTB` so you may not be able to use them yourself
  - You might see the LEDs on these pins flash during programming and debugging
  - If you haven't done so already, change your code to use port C instead for output.

Running the in circuit programmer/debugger

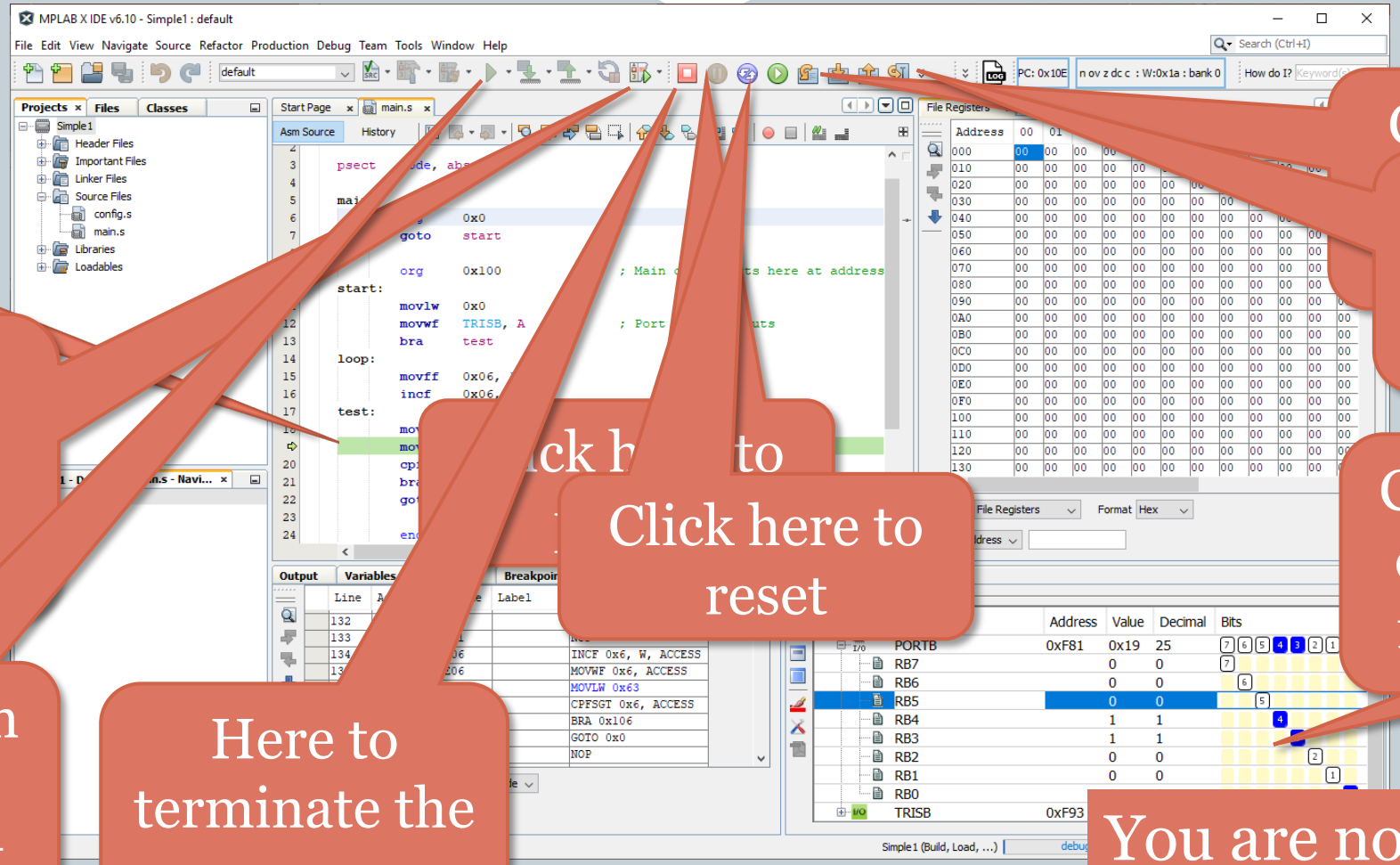


Select ICD by its type and serial number

Right click on project Simple1 to find and open project properties

Click OK to save and close

Stepping through with the ICD



Programs to write I:

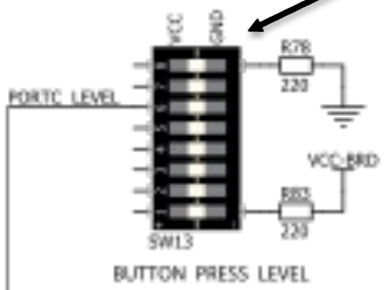
- Download your program to the board and run it.
  - What do you see on the LEDs?
- Modify the program so that it changes the number of times the loop is run depending on which switch button is pushed
  - Set the on-board dip-switch to connect the push buttons to +5v for port D, set `TRISD` to all ones to make it an input.
  - Leave the LED dip-switch in the connected position to pull inputs low when the button is not pressed
  - Look in file register, eg `PORTD`, to get the state of the buttons on ports

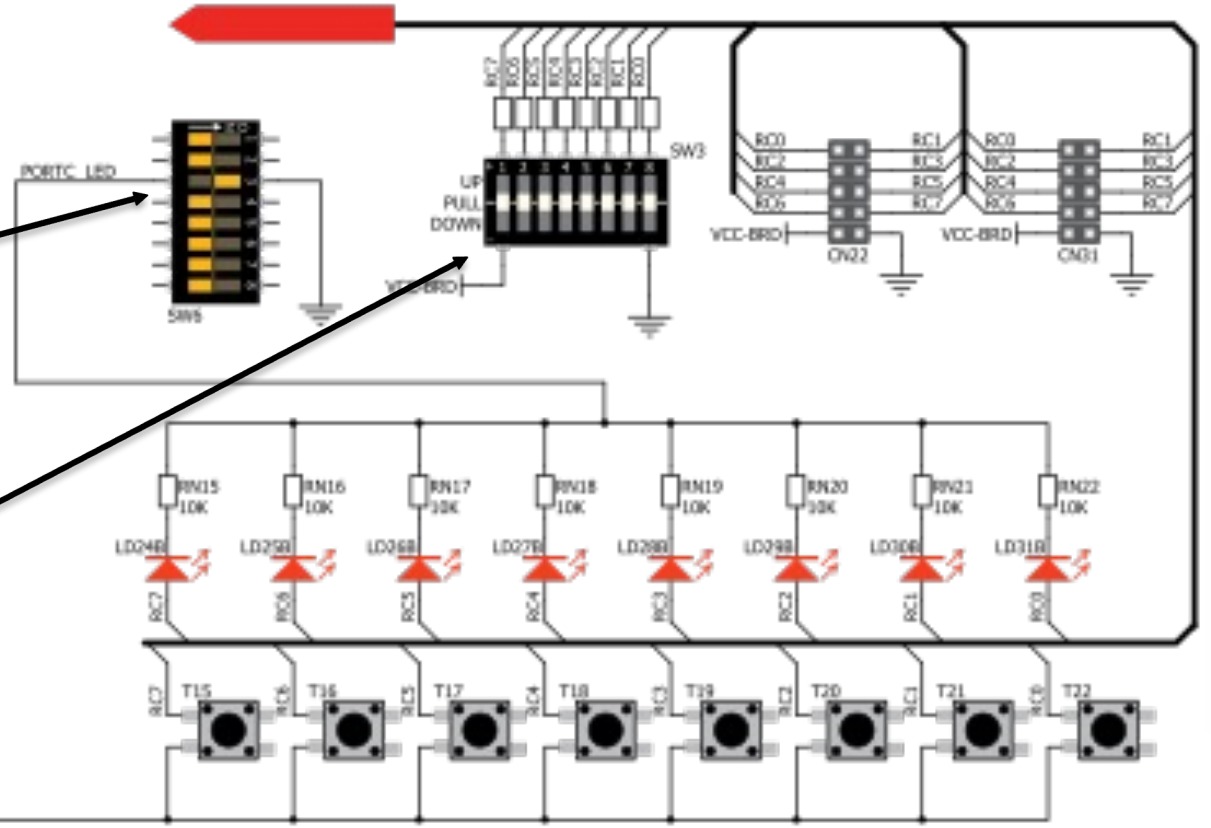
Board circuit around microprocessor ports

All ports have both LEDs and Push buttons attached

You can set the voltage at the other end of the LEDs and switches for each port

Or even the voltage on each port pin directly (leave this in "tri-state" position for now).





Looking at PORTD pins, and memory banking

- Consider the following replacements for the `movlw 0x63` instruction that sets the loop length:

```
1.  movf   PORTD, W, A
2.  movf   PORTD, W, B
3.  movlb  0xf
    movf   PORTD, W, B
4.  movf   PORTD, W
```

- 1, 3 and 4 should work!
- Note that the default for 4 is to assume "`ACCESS`" as `PORTD` is in the access ram area – but you might see a warning!
- If you try to address a file register outside access ram then "`BANKED`" is assumed, but you must set the `BSR` appropriately beforehand

Programs to write II:

- Make your counter count from 0 – FF, output the values to PORTC and look with your scope probe at the LSB (`RC0`).
  - How long does it take to make an addition?
  - Why does the `RC0` bit toggle with a frequency that is twice that of `RC1`? *(check this using two scope probes; one on `RC0` and another on `RC1`)*
- In the documentation you will find how many clock counts are required to perform each instruction in your program
  - The PIC should be running off a 64 MHz clock, but takes multiples of 4 clock cycles per instruction.
  - Predict the time it takes to compute your addition loop and compare with your measurement using the scope.
- MPLAB-X also has a "logic analyser" window in the simulator that will let you watch the simulated voltage level on pins as the program runs

Things to do at home…

- You can install MPLAB-X on your own computer
  - Its free, and it runs on Windows, Mac and Linux
  - Check out what you can do with the debugger and simulator
- Check out what Git can do for you
  - Lots of online material available
  - See Imperial ICT web pages about Git and GitHub
  - Sign up for a Github account and link it to the Imperial College enterprise account
  - Look to create your own “Fork” of the Microprocessors Lab repository so that you can store your work online too
  - Checkout the Git Tricks and Tips powerpoint on Blackboard