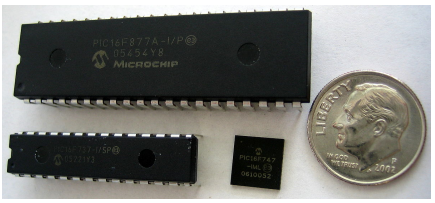
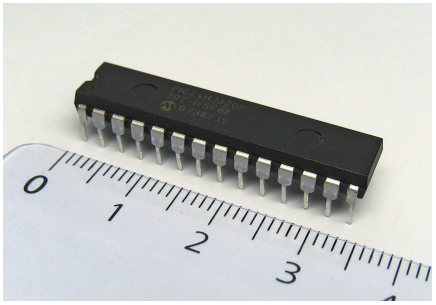
PIC microcontroller

PIC is a family of modified Harvard architecture microcontrollers made by Microchip Technology, derived from the PIC1650[1][2][3] originally developed by General Instrument's Microelectronics Division. The name PIC initially referred to "**Peripheral Interface Controller**".[4][5]

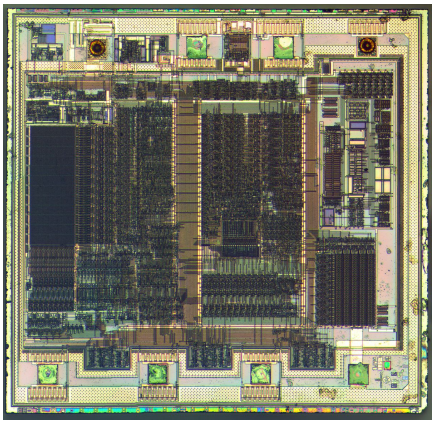
PICs are popular with both industrial developers and hobbyists alike due to their low cost, wide availability, large user base, extensive collection of application notes, availability of low cost or free development tools, and serial programming (and re-programming with flash memory) capability. They are also commonly used in educational programming as they often come with the easy to use 'pic logicator' software.



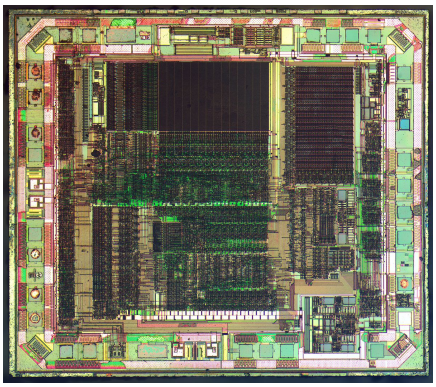
PIC microcontrollers in DIP and QFN packages



16-bit 28-pin PDIP PIC24 microcontroller next to a metric ruler

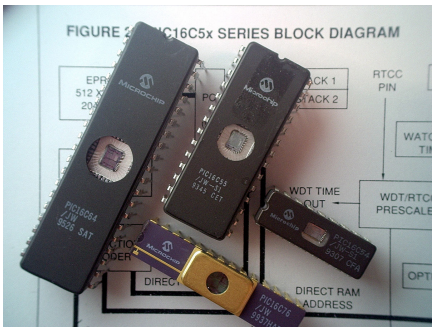


Die of a PIC12C508 8-bit, fully static, EEPROM/EPROM/ROM-based CMOS microcontroller manufactured by Microchip Technology using a 1200 nanometre process.



Die of a PIC16C505 CMOS ROM-based 8-bit microcontroller manufactured by Microchip Technology using a 1200 nanometre process.

History



Various older (EPROM) PIC microcontrollers

The original PIC was built to be used with General Instrument's new 16-bit CPU, the CP1600. While generally a good CPU, the CP1600 had poor I/O performance, and the 8-bit PIC was developed in 1975 to improve performance of the overall system by offloading I/O tasks from the CPU. The PIC used simple microcode stored in ROM to perform its tasks, and although the term was not used at the time, it shares some common features with RISC designs.

In 1985, General Instrument spun off their microelectronics division and the new ownership cancelled almost everything — which by this time was mostly out-of-date. The PIC, however, was upgraded with internal EPROM to produce a programmable channel controller and today a huge variety of PICs are available with various on-board peripherals (serial communication modules, UARTs, motor control kernels, etc.) and program memory from 256 words to 64k words and more (a "word" is one assembly language instruction, varying from 12, 14 or 16 bits depending on the specific PIC micro family).

PIC and PICmicro are registered trademarks of Microchip Technology. It is generally thought that PIC stands for **Peripheral Interface Controller**, although General Instruments' original acronym for the initial PIC1640 and PIC1650 devices was "**Programmable Interface Controller**".[4] The acronym was quickly replaced with "**Programmable Intelligent Computer**".[5]

The Microchip 16C84 (PIC16x84), introduced in 1993, was the first[*citation needed*] Microchip CPU with on-chip EEPROM memory. This electrically erasable memory made it cost less than CPUs that required a quartz "erase window" for erasing EPROM.

Core architecture

The PIC architecture is characterized by its multiple attributes:

- Separate code and data spaces (Harvard architecture).
- A small number of fixed length instructions
- Most instructions are single cycle execution (2 clock cycles, or 4 clock cycles in 8-bit models), with one delay cycle on branches and skips
- One accumulator (W0), the use of which (as source operand) is implied (i.e. is not encoded in the opcode)
- All RAM locations function as registers as both source and/or destination of math and other functions.[6]
- A hardware stack for storing return addresses
- A small amount of addressable data space (32, 128, or 256 bytes, depending on the family), extended through banking
- Data space mapped CPU, port, and peripheral registers
- ALU status flags are mapped into the data space
- The program counter is also mapped into the data space and writable (this is used to implement indirect jumps).

There is no distinction between memory space and register space because the RAM serves the job of both memory and registers, and the RAM is usually just referred to as the register file or simply as the registers.

Data space (RAM)

PICs have a set of registers that function as general purpose RAM. Special purpose control registers for on-chip hardware resources are also mapped into the data space. The addressability of memory varies depending on device series, and all PIC devices have some banking mechanism to extend addressing to additional memory. Later series of devices feature move instructions which can cover the whole addressable space, independent of the selected bank. In earlier devices, any register move had to be achieved via the accumulator.

To implement indirect addressing, a "file select register" (FSR) and "indirect register" (INDF) are used. A register number is written to the FSR, after which reads from or writes to INDF will actually be to or from the register pointed to by FSR. Later devices extended this concept with post- and pre- increment/decrement for greater efficiency in accessing sequentially stored data. This also allows FSR to be treated almost like a stack pointer (SP).

External data memory is not directly addressable except in some high pin count PIC18 devices.

Code space

The code space is generally implemented as ROM, EPROM or flash ROM. In general, external code memory is not directly addressable due to the lack of an external memory interface. The exceptions are PIC17 and select high pin count PIC18 devices.[7]

Word size

All PICs handle (and address) data in 8-bit chunks. However, the unit of addressability of the code space is not generally the same as the data space. For example, PICs in the baseline (PIC12) and mid-range (PIC16) families have program memory addressable in the same wordsize as the instruction width, i.e. 12 or 14 bits respectively. In contrast, in the PIC18 series, the program memory is addressed in 8-bit increments (bytes), which differs from the instruction width of 16 bits.

In order to be clear, the program memory capacity is usually stated in number of (single word) instructions, rather than in bytes.

Stacks

PICs have a hardware call stack, which is used to save return addresses. The hardware stack is not software accessible on earlier devices, but this changed with the 18 series devices.

Hardware support for a general purpose parameter stack was lacking in early series, but this greatly improved in the 18 series, making the 18 series architecture more friendly to high level language compilers.

Instruction set

A PIC's instructions vary from about 35 instructions for the low-end PICs to over 80 instructions for the high-end PICs. The instruction set includes instructions to perform a variety of operations on registers directly, the accumulator and a literal constant or the accumulator and a register, as well as for conditional execution, and program branching.

Some operations, such as bit setting and testing, can be performed on any numbered register, but bi-operand arithmetic operations always involve W (the accumulator), writing the result back to either W or the other operand register. To load a constant, it is necessary to load it into W before it can be moved into another register. On the older cores, all register moves needed to pass through W, but this changed on the "high end" cores.

PIC cores have skip instructions which are used for conditional execution and branching. The skip instructions are 'skip if bit set' and 'skip if bit not set'. Because cores before PIC18 had only unconditional branch instructions, conditional jumps are implemented by a conditional skip (with the opposite condition) followed by an unconditional branch. Skips are also of utility for conditional execution of any immediate single following instruction. It is possible to skip skip instructions. For example, the instruction sequence "skip if A; skip if B; C" will execute C if A is true or if B is false.

The 18 series implemented shadow registers which save several important registers during an interrupt, providing hardware support for automatically saving processor state when servicing interrupts.

In general, PIC instructions fall into 5 classes:

1. Operation on working register (WREG) with 8-bit immediate ("literal") operand. E.g. `movlw` (move literal to WREG), `andlw` (AND literal with WREG). One instruction peculiar to the PIC is `retlw`, load immediate into WREG and return, which is used with computed branches to produce lookup tables.
2. Operation with WREG and indexed register. The result can be written to either the Working register (e.g. `addwf` *`reg`*`,w`). or the selected register (e.g. `addwf` *`reg`*`,f`).
3. Bit operations. These take a register number and a bit number, and perform one of 4 actions: set or clear a bit, and test and skip on set/clear. The latter are used to perform conditional branches. The usual ALU status flags are available in a numbered register so operations such as "branch on carry clear" are possible.
4. Control transfers. Other than the skip instructions previously mentioned, there are only two: `goto` and `call`.
5. A few miscellaneous zero-operand instructions, such as return from subroutine, and `sleep` to enter low-power mode.

Performance

The architectural decisions are directed at the maximization of speed-to-cost ratio. The PIC architecture was among the first scalar CPU designs,[*citation needed*] and is still among the simplest and cheapest. The Harvard architecture—in which instructions and data come from separate sources—simplifies timing and microcircuit design greatly, and this benefits clock speed, price, and power consumption.

The PIC instruction set is suited to implementation of fast lookup tables in the program space. Such lookups take one instruction and two instruction cycles. Many functions can be modeled in this way. Optimization is facilitated by the relatively large program space of the PIC (e.g. 4096 × 14-bit words on the 16F690) and by the design of the instruction set, which allows for embedded constants. For example, a branch instruction's target may be indexed by W, and execute a "RETLW" which does as it is named - return with literal in W.

Interrupt latency is constant at three instruction cycles. External interrupts have to be synchronized with the four clock instruction cycle, otherwise there can be a one instruction cycle jitter. Internal interrupts are already synchronized. The constant interrupt latency allows PICs to achieve interrupt driven low jitter timing sequences. An example of this is a video sync pulse generator. This is no longer true in the newest PIC models, because they have a synchronous interrupt latency of three or four cycles.

Advantages

- Small instruction set to learn
- RISC architecture
- Built in oscillator with selectable speeds
- Easy entry level, in circuit programming plus in circuit debugging PICKit units available for less than $50
- Inexpensive microcontrollers
- Wide range of interfaces including I²C, SPI, USB, USART, A/D, programmable comparators, PWM, LIN, CAN, PSP, and Ethernet[8]
- Availability of processors in DIL package make them easy to handle for hobby use.

Limitations

- One accumulator
- Register-bank switching is required to access the entire RAM of many devices
- Operations and registers are not orthogonal; some instructions can address RAM and/or immediate constants, while others can only use the accumulator

The following stack limitations have been addressed in the **PIC18** series, but still apply to earlier cores:

- The hardware call stack is not addressable, so preemptive task switching cannot be implemented
- Software-implemented stacks are not efficient, so it is difficult to generate reentrant code and support local variables

With paged program memory, there are two page sizes to worry about: one for CALL and GOTO and another for computed GOTO (typically used for table lookups). For example, on PIC16, CALL and GOTO have 11 bits of addressing, so the page size is 2048 instruction words. For computed GOTOs, where you add to PCL, the page size is 256 instruction words. In both cases, the upper address bits are provided by the PCLATH register. This register must be changed every time control transfers between pages. PCLATH must also be preserved by any interrupt handler.[9]

Compiler development

While several commercial compilers are available, in 2008, Microchip released their own C compilers, C18 and C30, for the line of 18F 24F and 30/33F processors.

The easy to learn RISC instruction set of the PIC assembly language code can make the overall flow difficult to comprehend. Judicious use of simple macros can increase the readability of PIC assembly language. For example, the original Parallax PIC assembler ("SPASM") has macros which hide W and make the PIC look like a two-address machine. It has macro instructions like "`mov b, a`" (move the data from address *a* to address *b*) and "`add b, a`" (add data from address *a* to data in address *b*). It also hides the skip instructions by providing three operand branch macro instructions such as "`cjne a, b, dest`" (compare *a* with *b* and jump to *dest* if they are not equal).

Family core architectural differences

PICmicro chips have a Harvard architecture, and instruction words are unusual sizes. Originally, 12-bit instructions included 5 address bits to specify the memory operand, and 9-bit branch destinations. Later revisions added opcode bits, allowing additional address bits.

Baseline core devices (12 bit)

These devices feature a 12-bit wide code memory, a 32-byte register file, and a tiny two level deep call stack. They are represented by the PIC10 series, as well as by some PIC12 and PIC16 devices. Baseline devices are available in 6-pin to 40-pin packages.

Generally the first 7 to 9 bytes of the register file are special-purpose registers, and the remaining bytes are general purpose RAM. Pointers are implemented using a register pair: after writing an address to the FSR (file select register), the INDF (indirect f) register becomes an alias for the addressed register. If banked RAM is implemented, the bank number is selected by the high 3 bits of the FSR. This affects register numbers 16–31; registers 0–15 are global and not affected by the bank select bits.

Because of the very limited register space (5 bits), 4 rarely read registers were not assigned addresses, but written by special instructions (`OPTION` and `TRIS`).

The ROM address space is 512 words (12 bits each), which may be extended to 2048 words by banking. `CALL` and `GOTO` instructions specify the low 9 bits of the new code location; additional high-order bits are taken from the status register. Note that a CALL instruction only includes 8 bits of address, and may only specify addresses in the first half of each 512-word page.

Lookup tables are implemented using a computed `GOTO` (assignment to PCL register) into a table of `RETLW` instructions.

The instruction set is as follows. Register numbers are referred to as "f", while constants are referred to as "k". Bit numbers (0–7) are selected by "b". The "d" bit selects the destination: 0 indicates W, while 1 indicates that the result is written back to source register f. The C and Z status flags may be set based on the result; otherwise they are unmodified. Add and subtract (but not rotate) instructions that set C also set the DC (digit carry) flag, the carry from bit 3 to bit 4, which is useful for BCD arithmetic.

12-bit PIC instruction set

11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	C?	Z?	Description
0	**0**	**0**	**0**	**0**	**0**	**0**	**opcode**					**Miscellaneous instructions**			
0	0	0	0	0	0	0	0	0	0	0	0	NOP			No operation (MOVW 0,W)
0	0	0	0	0	0	0	0	0	0	1	0	OPTION			Copy W to OPTION register
0	0	0	0	0	0	0	0	0	0	1	1	SLEEP			Go into standby mode
0	0	0	0	0	0	0	0	0	1	0	0	CLRWDT			Restart watchdog timer
0	0	0	0	0	0	0	0	0	1	*f*		TRIS *f*			Copy W to tri-state register (*f* = 1, 2 or 3)
0	**0**	***opcode***				***d***	***register***					**ALU operations: dest ← OP(f,W)**			
0	0	0	0	0	0	1	*f*					MOVWF *f*			dest ← W
0	0	0	0	0	1	*d*	*f*					CLR *f*,*d*		Z	dest ← 0, usually written CLRW or CLRF *f*
0	0	0	0	1	0	*d*	*f*					SUBWF *f*,*d*	C	Z	dest ← f–W (dest ← f+~W+1)
0	0	0	0	1	1	*d*	*f*					DECF *f*,*d*		Z	dest ← f–1
0	0	0	1	0	0	*d*	*f*					IORWF *f*,*d*		Z	dest ← f | W, logical inclusive or
0	0	0	1	0	1	*d*	*f*					ANDWF *f*,*d*		Z	dest ← f & W, logical and
0	0	0	1	1	0	*d*	*f*					XORWF *f*,*d*		Z	dest ← f ^ W, logical exclusive or
0	0	0	1	1	1	*d*	*f*					ADDWF *f*,*d*	C	Z	dest ← f+W
0	0	1	0	0	0	*d*	*f*					MOVF *f*,*d*		Z	dest ← f
0	0	1	0	0	1	*d*	*f*					COMF *f*,*d*		Z	dest ← ~f, bitwise complement
0	0	1	0	1	0	*d*	*f*					INCF *f*,*d*		Z	dest ← f+1
0	0	1	0	1	1	*d*	*f*					DECFSZ *f*,*d*			dest ← f–1, then skip if zero
0	0	1	1	0	0	*d*	*f*					RRF *f*,*d*	C		dest ← CARRY<<7 | f>>1, rotate right through carry
0	0	1	1	0	1	*d*	*f*					RLF *f*,*d*	C		dest ← F<<1 | CARRY, rotate left through carry
0	0	1	1	1	0	*d*	*f*					SWAPF *f*,*d*			dest ← f<<4 | f>>4, swap nibbles
0	0	1	1	1	1	*d*	*f*					INCFSZ *f*,*d*			dest ← f+1, then skip if zero
0	**1**	***op***		***bit***			***register***					**Bit operations**			
0	1	0	0	*bit*			*f*					BCF *f*,*b*			Clear bit b of f
0	1	0	1	*bit*			*f*					BSF *f*,*b*			Set bit b of f
0	1	1	0	*bit*			*f*					BTFSC *f*,*b*			Skip if bit b of f is clear
0	1	1	1	*bit*			*f*					BTFSS *f*,*b*			Skip if bit b of f is set
1	**0**	***op***		***k***								**Control transfers**			
1	0	0	0	*k*								RETLW *k*			Set W ← k, then return from subroutine
1	0	0	1	*k*								CALL *k*			Call subroutine, 8-bit address k
1	0	1	*k*									GOTO *k*			Jump to 9-bit address k[10]
1	**1**	***op***		***8-bit immediate***								**Operations with W and 8-bit literal: W ← OP(k,W)**			
1	1	0	0	*k*								MOVLW *k*			W ← k
1	1	0	1	*k*								IORLW *k*		Z	W ← k | W, bitwise logical or

1	1	1	0	*k*								ANDLW *k*		Z	W ← k & W, bitwise and
1	1	1	1	*k*								XORLW *k*		Z	W ← k ^ W, bitwise exclusive or
11	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Mnemonic**	**C?**	**Z?**	**Description**

ELAN Microelectronics clones (13 bit)

ELAN Microelectronics Corp. make a series of PICmicro-like microcontrollers with a 13-bit instruction word.[11] The instructions are mostly compatible with the mid-range 14-bit instruction set, but limited to a 6-bit register address (16 special-purpose registers and 48 bytes of RAM) and a 10-bit (1024 word) program space.

The 10-bit program counter is accessible as R2. Reads access only the low bits, and writes clear the high bits. An exception is the TBL instruction, which modifies the low byte while preserving bits 8 and 9.

The 7 accumulator-immediate instructions are renumbered relative to the 14-bit PICmicro, to fit into 3 opcode bits rather than 4, but they are all there, as well as an additional software interrupt instruction.

There are a few additional miscellaneous instructions, and there are some changes to the terminology (the PICmicro OPTION register is called the CONTrol register; the PICmicro TRIS registers 1–3 are called I/O control registers 5–7), but the equivalents are obvious.

13-bit EM78 instruction set[12]

12	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Mnemonic**	**C?**	**Z?**	**Description**
0	**0**	**0**	**0**	**0**	**0**	**0**	**opcode**						**Miscellaneous instructions**			
0	0	0	0	0	0	0	0	0	0	0	0	0	NOP*			No operation (MOVW 0,W)
0	0	0	0	0	0	0	0	0	0	0	0	1	DAA†	C		Decimal Adjust after Addition
0	0	0	0	0	0	0	0	0	0	0	1	0	CONTW*			Write CONT register (CONT ← W)
0	0	0	0	0	0	0	0	0	0	0	1	1	SLEEP*			Go into standby mode (WDT ← 0, stop clock)
0	0	0	0	0	0	0	0	0	0	1	0	0	CLRWDT*			Restart watchdog timer (WDT ← 0)
0	0	0	0	0	0	0	0	0	*f*				IOW *f**			Copy W to I/O control register (*f* = 5–7, 11–15)
0	0	0	0	0	0	0	0	1	0	0	0	0	ENI†			Enable interrupts
0	0	0	0	0	0	0	0	1	0	0	0	1	DISI†			Disable interrupts
0	0	0	0	0	0	0	0	1	0	0	1	0	RET			Return from subroutine, W unmodified
0	0	0	0	0	0	0	0	1	0	0	1	1	RETI			Return from interrupt; return & enable interrupts
0	0	0	0	0	0	0	0	1	0	1	0	0	CONTR†			Read CONT register (W ← CONT)
0	0	0	0	0	0	0	0	1	*f*				IOR *f*‡			Copy I/O control register to W (*f* = 5–7, 11–15)
0	0	0	0	0	0	0	1	0	0	0	0	0	TBL†	C	Z	PCL += W. preserve PC bits 8 & 9
0	**0**	***opcode***				***d***	***register***						**ALU operations same as 12- and 14-bit PIC**			
0	**1**	***op***		***bit***			***register***						**Bit operations same as 12- and 14-bit PIC**			
1	**0**	***op***	***k***										**Control transfers same as 14-bit PIC**			
1	**1**	***opcode***			***8-bit immediate***								**Operations with W and 8-bit literal: W ← OP(k,W)**			

1	1	0	*op*		*k*	MOV/IOR/AND/XOR, same as 12-bit PIC			
1	1	1	0	0	*k*	RETLW *k*			W ← k, then return from subroutine
1	1	1	0	1	*k*	SUBLW *k*	C	Z	W ← k–W (W ← k+~W+1)
1	1	1	1	0	*k*	INT *k*†			Push PC, PC ← k (software interrupt, usually k=1)
1	1	1	1	1	*k*	ADDLW *k*	C	Z	W ← k+W

12	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Mnemonic**	**C?**	**Z?**	**Description**

*: Same opcode as 12-bit PIC

†: Instruction unique to EM78 instruction set with no PIC equivalent

Some models support multiple ROM or RAM banks, in a manner similar to other PIC microcontrollers.

Mid-range core devices (14 bit)

These devices feature a 14-bit wide code memory, and an improved 8 level deep call stack. The instruction set differs very little from the baseline devices, but the 2 additional opcode bits allow 128 registers and 2048 words of code to be directly addressed. There are a few additional miscellaneous instructions, and two additional 8-bit literal instructions, add and subtract. The mid-range core is available in the majority of devices labeled PIC12 and PIC16.

The first 32 bytes of the register space are allocated to special-purpose registers; the remaining 96 bytes are used for general-purpose RAM. If banked RAM is used, the high 16 registers (0x70–0x7F) are global, as are a few of the most important special-purpose registers, including the STATUS register which holds the RAM bank select bits. (The other global registers are FSR and INDF, the low 8 bits of the program counter PCL, the PC high preload register PCLATH, and the master interrupt control register INTCON.)

The PCLATH register supplies high-order instruction address bits when the 8 bits supplied by a write to the PCL register, or the 11 bits supplied by a `GOTO` or `CALL` instruction, is not sufficient to address the available ROM space.

14-bit PIC instruction set

13	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Mnemonic**	**C?**	**Z?**	**Description**
0	**0**	**0**	**0**	**0**	**0**	**0**	**opcode**							**Miscellaneous instructions**			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	NOP			No operation (MOVW 0,W)
0	0	0	0	0	0	0	0	0	0	1	0	0	0	RETURN			Return from subroutine, W unmodified
0	0	0	0	0	0	0	0	0	0	1	0	0	1	RETFIE			Return from interrupt
0	0	0	0	0	0	0	1	1	0	0	0	1	0	OPTION			Copy W to OPTION register
0	0	0	0	0	0	0	1	1	0	0	0	1	1	SLEEP			Go into standby mode
0	0	0	0	0	0	0	1	1	0	0	1	0	0	CLRWDT			Restart watchdog timer
0	0	0	0	0	0	0	1	1	0	0	1	*f*		TRIS *f*			Copy W to tri-state register (*f* = 1, 2 or 3)
0	**0**	***opcode***				***d***	***register***							**ALU operations: dest ← OP(f,W)**			
0	0	0	0	0	0	1	*f*							MOVWF *f*			f ← W
0	0	0	0	0	1	*d*	*f*							CLR *f*,*d*		Z	dest ← 0, usually written CLRW or CLRF *f*
0	0	0	0	1	0	*d*	*f*							SUBWF *f*,*d*	C	Z	dest ← f–W (dest ← f+~W+1)
0	0	0	0	1	1	*d*	*f*							DECF *f*,*d*		Z	dest ← f–1
0	0	0	1	0	0	*d*	*f*							IORWF *f*,*d*		Z	dest ← f | W, logical inclusive or

0	0	0	1	0	1	*d*	*f*	ANDWF *f*,*d*		Z	dest ← f & W, logical and
0	0	0	1	1	0	*d*	*f*	XORWF *f*,*d*		Z	dest ← f ^ W, logical exclusive or
0	0	0	1	1	1	*d*	*f*	ADDWF *f*,*d*	C	Z	dest ← f+W
0	0	1	0	0	0	*d*	*f*	MOVF *f*,*d*		Z	dest ← f
0	0	1	0	0	1	*d*	*f*	COMF *f*,*d*		Z	dest ← ~f, bitwise complement
0	0	1	0	1	0	*d*	*f*	INCF *f*,*d*		Z	dest ← f+1
0	0	1	0	1	1	*d*	*f*	DECFSZ *f*,*d*			dest ← f−1, then skip if zero
0	0	1	1	0	0	*d*	*f*	RRF *f*,*d*	C		dest ← CARRY<<7 | f>>1, rotate right through carry
0	0	1	1	0	1	*d*	*f*	RLF *f*,*d*	C		dest ← f<<1 | CARRY, rotate left through carry
0	0	1	1	1	0	*d*	*f*	SWAPF *f*,*d*			dest ← f<<4 | f>>4, swap nibbles
0	0	1	1	1	1	*d*	*f*	INCFSZ *f*,*d*			dest ← f+1, then skip if zero

0	**1**	***op***		***bit***	***register***	**Bit operations**			
0	1	0	0	*bit*	*f*	BCF *f*,*b*			Clear bit b of f
0	1	0	1	*bit*	*f*	BSF *f*,*b*			Set bit b of f
0	1	1	0	*bit*	*f*	BTFSC *f*,*b*			Skip if bit b of f is clear
0	1	1	1	*bit*	*f*	BTFSS *f*,*b*			Skip if bit b of f is set

1	**0**	***op***	***k***	**Control transfers**			
1	0	0	*k*	CALL *k*			Call subroutine
1	0	1	*k*	GOTO *k*			Jump to address k

1	**1**	***opcode***				***8-bit immediate***	**Operations with W and 8-bit literal: W ← OP(k,W)**			
1	1	0	0	*x*	*x*	*k*	MOVLW *k*			W ← k
1	1	0	1	*x*	*x*	*k*	RETLW *k*			W ← k, then return from subroutine
1	1	1	0	0	0	*k*	IORLW *k*		Z	W ← k | W, bitwise logical or
1	1	1	0	0	1	*k*	ANDLW *k*		Z	W ← k & W, bitwise and
1	1	1	0	1	0	*k*	XORLW *k*		Z	W ← k ^ W, bitwise exclusive or
1	1	1	0	1	1	*k*	*(reserved)*			
1	1	1	1	0	*x*	*k*	SUBLW *k*	C	Z	W ← k−W (dest ← k+~W+1)
1	1	1	1	1	*x*	*k*	ADDLW *k*	C	Z	W ← k+W

13	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Mnemonic**	**C?**	**Z?**	**Description**

Enhanced mid-range core devices (14 bit)

Enhanced mid-range core devices introduce a deeper hardware stack, additional reset methods, 14 additional instructions and 'C' programming language optimizations. In particular. there are two `INDF registers` (`INDF0` and `INDF1`), and two corresponding `FSR` register pairs (`FSR`*n*`L` and `FSR`*n*`H`). Special instructions use `FSR`*n* registers like address registers, with a variety of addressing modes.

14-bit enhanced PIC additional instructions

13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	C?	Z?	Description
0	**0**	**0**	**0**	**0**	**0**	**0**	**opcode**							**Miscellaneous instructions**			
0	0	0	0	0	0	0	0	0	0	0	0	0	1	RESET			Software reset
0	0	0	0	0	0	0	0	0	0	1	0	1	0	CALLW			Push PC, then jump to PCLATH:W
0	0	0	0	0	0	0	0	0	0	1	0	1	1	BRW			PC ← PC + W, relative jump using W
0	0	0	0	0	0	0	0	0	1	0	*n*	0	0	MOVIW ++FSR*n*		Z	Increment FSR*n*, then W ← INDF*n*
0	0	0	0	0	0	0	0	0	1	0	*n*	0	1	MOVIW −−FSR*n*		Z	Decrement FSR*n*, then W ← INDF*n*
0	0	0	0	0	0	0	0	0	1	0	*n*	1	0	MOVIW FSR*n*++		Z	W ← INDF*n*, then increment FSR*n*
0	0	0	0	0	0	0	0	0	1	0	*n*	1	1	MOVIW FSR*n*−−		Z	W ← INDF*n*, then decrement FSR*n*
0	0	0	0	0	0	0	0	0	1	1	*n*	*m*		MOVWI using FSR*n*			INDF*n* ← W, same modes as MOVIW
0	0	0	0	0	0	0	0	1	*k*					MOVLB *k*			BSR ← k, move literal to bank select register
1	**1**	***opcode***				***d***	***register***							**ALU operations: dest ← OP(f,W)**			
1	1	0	1	0	1	*d*	*f*							LSLF *f*,*d*	C	Z	dest ← *f* << 1, logical shift left
1	1	0	1	1	0	*d*	*f*							LSRF *f*,*d*	C	Z	dest ← *f* >> 1, logical shift right
1	1	0	1	1	1	*d*	*f*							ASRF *f*,*d*	C	Z	dest ← *f* >> 1, arithmetic shift right
1	1	1	0	1	1	*d*	*f*							SUBWFB *f*,*d*	C	Z	dest ← *f* + ~W + C, subtract with carry
1	1	1	1	0	1	*d*	*f*							ADDWFC *f*,*d*	C	Z	dest ← *f* + W + C, add with carry
1	**1**	***opcode***				***k***								**Operations with literal *k***			
1	1	0	0	0	1	0	*n*	*k*						ADDFSR FSR*n*,*k*			FSR*n* ← FSR*n* + k, add 6-bit signed offset
1	1	0	0	0	1	1	*k*							MOVLP *k*			PCLATH ← k, move 7-bit literal to PC latch high
1	1	0	0	1	*k*									BRA *k*			PC ← PC + k, branch relative using 9-bit signed offset
1	1	1	1	1	1	0	*n*	*k*						MOVIW *k*[FSR*n*]		Z	W ← [FSR*n*+*k*], 6-bit signed offset
1	1	1	1	1	1	1	*n*	*k*						MOVWI *k*[FSR*n*]			[FSR*n*+*k*] ← W, 6-bit signed offset
13	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Mnemonic**	**C?**	**Z?**	**Description**

PIC17 high end core devices (16 bit)

The 17 series never became popular and has been superseded by the PIC18 architecture. It is not recommended for new designs, and availability may be limited.

Improvements over earlier cores are 16-bit wide opcodes (allowing many new instructions), and a 16 level deep call stack. PIC17 devices were produced in packages from 40 to 68 pins.

The 17 series introduced a number of important new features:

- a memory mapped accumulator
- read access to code memory (table reads)
- direct register to register moves (prior cores needed to move registers through the accumulator)
- an external program memory interface to expand the code space
- an 8-bit × 8-bit hardware multiplier
- a second indirect register pair
- auto-increment/decrement addressing controlled by control bits in a status register (ALUSTA)

PIC18 high end core devices (16 bit)

Microchip introduced the PIC18 architecture in 2000. [13] Unlike the 17 series, it has proven to be very popular, with a large number of device variants presently in manufacture. In contrast to earlier devices, which were more often than not programmed in assembly, C has become the predominant development language.[14]

The 18 series inherits most of the features and instructions of the 17 series, while adding a number of important new features:

- call stack is 21 bits wide and much deeper (31 levels deep)
- the call stack may be read and written (TOSU:TOSH:TOSL registers)
- conditional branch instructions
- indexed addressing mode (PLUSW)
- extending the FSR registers to 12 bits, allowing them to linearly address the entire data address space
- the addition of another FSR register (bringing the number up to 3)

The RAM space is 12 bits, addressed using a 4-bit bank select register and an 8-bit offset in each instruction. An additional "access" bit in each instruction selects between bank 0 (*a*=0) and the bank selected by the BSR (*a*=1).

A 1-level stack is also available for the STATUS, WREG and BSR registers. They are saved on every interrupt, and may be restored on return. If interrupts are disabled, they may also be used on subroutine call/return by setting the *s* bit (appending ", FAST" to the instruction).

The auto increment/decrement feature was improved by removing the control bits and adding four new indirect registers per FSR. Depending on which indirect file register is being accessed it is possible to postdecrement, postincrement, or preincrement FSR; or form the effective address by adding W to FSR.

In more advanced PIC18 devices, an "extended mode" is available which makes the addressing even more favorable to compiled code:

- a new offset addressing mode; some addresses which were relative to the access bank are now interpreted relative to the FSR2 register
- the addition of several new instructions, notable for manipulating the FSR registers.

These changes were primarily aimed at improving the efficiency of a data stack implementation. If FSR2 is used either as the stack pointer or frame pointer, stack items may be easily indexed—allowing more efficient re-entrant code. Microchip's MPLAB C18 C compiler chooses to use FSR2 as a frame pointer.

PIC18 16-bit instruction set[15]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Mnemonic	C?	Z?	N?	Description
0	**0**	**0**	**0**	**0**	**0**	**0**	**opcode**									**Miscellaneous instructions**				
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	NOP				No operation
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	SLEEP				Go into standby mode
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	CLRWDT				Restart watchdog timer
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	PUSH				Push PC on top of stack
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	POP				Pop (and discard) top of stack
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	DAW	C			Decimal adjust W
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	TBLRD*				Table read
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	TBLRD*+				Table read with postincrement
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	TBLRD*−				Table read with postdecrement
0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	TBLRD+*				Table read with pre-increment
0	0	0	0	0	0	0	0	0	0	0	0	1	1	*mod*		TBLWR				Table write, same modes as TBLRD
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	*s*	RETFIE [, FAST]				Return from interrupt
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	*s*	RETURN [, FAST]				Return from subroutine
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	RESET	cleared			Software reset
0	0	0	0	0	0	0	1	0	0	0	0	*k*				MOVLB				Move literal *k* to bank select register
0	***opcode***					***d***	***a***	***register***								**ALU operations: dest ← OP(f,W)**				
0	0	0	0	0	0	1	*a*	*f*								MULWF *f*,*a*				PRODH:PRODL ← W × f (unsigned)
0	0	0	0	0	1	*d*	*a*	*f*								DECF *f*,*d*,*a*	C	Z	N	dest ← f − 1
0	0	0	1	0	0	*d*	*a*	*f*								IORWF *f*,*d*,*a*		Z	N	dest ← f | W, logical inclusive or
0	0	0	1	0	1	*d*	*a*	*f*								ANDWF *f*,*d*,*a*		Z	N	dest ← f & W, logical and
0	0	0	1	1	0	*d*	*a*	*f*								XORWF *f*,*d*,*a*		Z	N	dest ← f ^ W, exclusive or
0	0	0	1	1	1	*d*	*a*	*f*								COMF *f*,*d*,*a*		Z	N	dest ← ~f, bitwise complement
0	0	1	0	0	0	*d*	*a*	*f*								ADDWFC *f*,*d*,*a*	C	Z	N	dest ← f + W + C
0	0	1	0	0	1	*d*	*a*	*f*								ADDWF *f*,*d*,*a*	C	Z	N	dest ← f + W
0	0	0	0	1	0	*d*	*a*	*f*								INCF *f*,*d*,*a*	C	Z	N	dest ← f + 1
0	0	1	0	1	1	*d*	*a*	*f*								DECFSZ *f*,*d*,*a*				dest ← f − 1, skip if 0
0	0	1	1	0	0	*d*	*a*	*f*								RRCF *f*,*d*,*a*	C	Z	N	dest ← f>>1 | C<<7, rotate right through carry
0	0	1	1	0	1	*d*	*a*	*f*								RLCF *f*,*d*,*a*	C	Z	N	dest ← f<<1 | C, rotate left through carry
0	0	1	1	1	0	*d*	*a*	*f*								SWAPF *f*,*d*,*a*				dest ← f<<4 | f>>4, swap nibbles
0	0	1	1	1	1	*d*	*a*	*f*								INCFSZ *f*,*d*,*a*				dest ← f + 1, skip if 0
0	1	0	0	0	0	*d*	*a*	*f*								RRNCF *f*,*d*,*a*		Z	N	dest ← f>>1 | f<<7, rotate right (no carry)
0	1	0	0	0	1	*d*	*a*	*f*								RLNCF *f*,*d*,*a*		Z	N	dest ← f<<1 | f>>7, rotate left (no carry)
0	1	0	0	1	0	*d*	*a*	*f*								INFSNZ *f*,*d*,*a*				dest ← f + 1, skip if not 0
0	1	0	0	1	1	*d*	*a*	*f*								DCFSNZ *f*,*d*,*a*				dest ← f − 1, skip if not 0
0	1	0	1	0	0	*d*	*a*	*f*								MOVF *f*,*d*,*a*		Z	N	dest ← f

0	1	0	1	0	1	*d*	*a*	*f*	SUBFWB *f*,*d*,*a*	C	Z	N	dest ← W + ~f + C (dest ← W − f − C̄)
0	1	0	1	1	0	*d*	*a*	*f*	SUBWFB *f*,*d*,*a*	C	Z	N	dest ← f + ~W + C (dest ← f − W − C̄)
0	1	0	1	1	1	*d*	*a*	*f*	SUBWF *f*,*d*,*a*	C	Z	N	dest ← f − W (dest ← f + ~W + 1)
0	**1**	**1**	**0**	***opcode***			***a***	***register***	**ALU operations, do not write to W**				
0	1	1	0	0	0	0	*a*	*f*	CPFSLT *f*,*a*				skip if f < W
0	1	1	0	0	0	1	*a*	*f*	CPFSEQ *f*,*a*				skip if f == W
0	1	1	0	0	1	0	*a*	*f*	CPFSGT *f*,*a*				skip if f > W
0	1	1	0	0	1	1	*a*	*f*	TSTFSZ *f*,*a*				skip if f == 0
0	1	1	0	1	0	0	*a*	*f*	SETF *f*,*a*				f ← 0xFF
0	1	1	0	1	0	1	*a*	*f*	CLRF *f*,*a*		1		f ← 0, PSR.Z ← 1
0	1	1	0	1	1	0	*a*	*f*	NEGF *f*,*a*	C	Z	N	f ← −f
0	1	1	0	1	1	1	*a*	*f*	MOVWF *f*,*a*				f ← W
1	**0**	***opc***		***bit***			***a***	***register***	**Bit operations**				
0	1	1	1	*bit*			*a*	*f*	BTG *f*,*b*,*a*				Toggle bit b of f
1	0	0	0	*bit*			*a*	*f*	BSF *f*,*b*,*a*				Set bit b of f
1	0	0	1	*bit*			*a*	*f*	BCF *f*,*b*,*a*				Clear bit b of f
1	0	1	0	*bit*			*a*	*f*	BTFSS *f*,*b*,*a*				Skip if bit b of f is set
1	0	1	1	*bit*			*a*	*f*	BTFSC *f*,*b*,*a*				Skip if bit b of f is clear
1	**1**	**0**	***opc***	***address***					**Large-address operations**				
1	1	0	0	*source*					MOVFF *s*,*d*				Move absolute
1	1	1	1	*destination*									
1	1	0	1	0	*n*				BRA *n*				Branch to PC + 2*n*
1	1	0	1	1	*n*				RCALL *n*				Subroutine call to PC + 2*n*
1	**1**	**1**	**0**	**0**	***cond***			***n***	**Conditional branch**				
1	1	1	0	0	0	0	0	*n*	BZ *n*				Branch if PSR.Z is set
1	1	1	0	0	0	0	1	*n*	BNZ *n*				Branch if PSR.Z is clear
1	1	1	0	0	0	1	0	*n*	BC *n*				Branch if PSR.C is set
1	1	1	0	0	0	1	1	*n*	BNC *n*				Branch if PSR.C is clear
1	1	1	0	0	1	0	0	*n*	BOV *n*				Branch if PSR.V is set
1	1	1	0	0	1	0	1	*n*	BNOV *n*				Branch if PSR.V is clear
1	1	1	0	0	1	1	0	*n*	BN *n*				Branch if PSR.N is set
1	1	1	0	0	1	1	1	*n*	BNN *n*				Branch if PSR.N is clear
1	1	1	0	1	0				*(reserved)*				
1	**1**	**1**	**0**	**1**	**1**	***opc***		***k***	**2-word instructions**				
1	1	1	0	1	1	0	*s*	*k* (lsbits)	CALL *k*[, FAST]				Call subroutine
1	1	1	1	*k* (msbits)									

1	1	1	0	1	1	1	0	0	0	*f*	*k* (msb)	LFSR *f*,*k*				Move 12-bit literal to FSR*f*
1	1	1	1	0	0	0	0	*k* (lsbits)								
1	1	1	0	1	1	1	1	*k* (lsbits)				GOTO *k*				Absolute jump, PC ← k
1	1	1	1	*k* (msbits)												
1	**1**	**1**	**1**	***k***								**(No operation)**				

PIC24 and dsPIC 16-bit microcontrollers

In 2001, Microchip introduced the dsPIC series of chips,[16] which entered mass production in late 2004. They are Microchip's first inherently 16-bit microcontrollers. PIC24 devices are designed as general purpose microcontrollers. dsPIC devices include digital signal processing capabilities in addition.

Although still similar to earlier PIC architectures, there are significant enhancements:[17]

- All registers are 16 bits wide
- Data address space expanded to 64 Kbytes
- First 2K is reserved for peripheral control registers
- Data bank switching is not required unless RAM exceeds 62K
- "f operand" direct addressing extended to 13 bits (8 Kbytes)
- 16 W registers available for register-register operations.

(But operations on f operands always reference W0.)

- Program counter is 22 bits (Bits 22:1; bit 0 is always 0)
- Instructions are 24 bits wide
- Instructions come in byte (B=1) and (16-bit) word (B=0) forms
- Stack is in RAM (with W15 as stack pointer); there is no hardware stack
- W14 is the frame pointer
- Data stored in ROM may be accessed directly ("Program Space Visibility")
- Interrupt vectors for different interrupt sources are supported.

Some features are:

- hardware MAC (multiply–accumulate)
- barrel shifting
- bit reversal
- (16×16)-bit single-cycle multiplication and other DSP operations
- hardware divide assist (19 cycles for 16/32-bit divide)
- hardware support for loop indexing
- Direct memory access

dsPICs can be programmed in C using Microchip's C30 compiler which is a variant of GCC.

Instruction ROM is 24 bits wide. Software can access ROM in 16-bit words, where even words hold the least significant 16 bits of each instruction, and odd words hold the most significant 8 bits. The high half of odd words reads as zero. The program counter is 23 bits wide, but the least significant bit is always 0, so there are 22 modifiable bits.

Instructions come in 2 main varieties. One is like the classic PIC instructions, with an operation between W0 and a value in a specified f register (i.e. the first 8K of RAM), and a destination select bit selecting which is updated with the result. The W registers are memory-mapped. so the f operand may be any W register,

The other form, new to the PIC24, specifies 3 W register operands, 2 of which allow a 3-bit addressing mode specification:

PIC24 addressing modes

s operand			d operand			Description
ppp	**Reg**	**Syntax**	**qqq**	**Reg**	**Syntax**	
000	ssss	Ws	000	dddd	Wd	Register direct
001	ssss	[Ws]	001	dddd	[Wd]	Indirect
010	ssss	[Ws−−]	010	dddd	[Wd−−]	Indirect with postdecrement
011	ssss	[Ws++]	011	dddd	[Wd++]	Indirect with postincrement
100	ssss	[−−Ws]	100	dddd	[−−Wd]	Indirect with predecrement
101	ssss	[++Ws]	101	dddd	[++Wd]	Indirect with preincrement
11*x*	ssss	[Ws+Ww]	11*x*	dddd	[Wd+Ww]	Indirect with register offset
11*k*	kkkk	#u5	(Unused, illegal)			5-bit unsigned immediate

The register offset addressing mode is only available to 2-operand instructions. 3-operand instructions use Ww as the second source operand, and use this encoding for an unsigned 5-bit immediate source. Note that the same Ww may be added to both Wd and Ws.

A few instructions are 2 words long. The second word is a NOP, which includes up to 16 bits of additional immediate operand.

PIC24 24-bit instruction set[]

23	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Mnemonic**	**C ?**	**Z ?**	**N ?**	**Description**
0	**0**	**0**	**0**	**opcode**				**offset**																**Control transfers**				
0	0	0	0	0	0	0	0	—																NOP				No operation (& 2nd instruction word)
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	n							NOP				Second word of absolute branch (bits 22:16)
0	0	0	0	0	0	0	0	n																NOP				Second word of DO instruction
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**op**		**—0—**							**a**						**Computed control transfer (to 16-bit Wa)**				
0	0	0	0	0	0	0	1	0	0	0	—0—							a						CALL Ra				Push PC, jump to Wa
0	0	0	0	0	0	0	1	0	0	1	—0—							a						RCALL Ra				Push PC, jump to PC+2×Wa
0	0	0	0	0	0	0	1	0	1	0	—0—							a						GOTO Ra				Jump to Wa
0	0	0	0	0	0	0	1	0	1	1	—0—							a						BRA Ra				Jump to PC+2×Wa
0	0	0	0	0	0	1	0	n<15:1>															0	CALL addr23				Push PC, jump to absolute address
—0—								0	0	0	0	0	0	0	0	0	n<22:16>											
0	0	0	0	0	0	1	1	—																(Reserved)				
0	0	0	0	0	1	0	0	n															0	GOTO addr23				Jump to absolute address
—0—								0	0	0	0	0	0	0	0	0	n<22:16>											
0	0	0	0	0	1	0	1	0	B	k								d						RETLW #k,Wd				Wn[.B] = #u10, pop PC
0	0	0	0	0	1	1	0	0	0	—0—														RETURN				pop PC
0	0	0	0	0	1	1	0	0	1	—0—														RETFIE	C	Z	N	pop SR, PC

0	0	0	0	0	1	1	1	n							RCALL address				Push PC, PC += 2*s16
0	0	0	0	1	0	0	0	0	0	k					DO #k, addr				Zero-overhead loop; k+1 is repeat count, PC+2n last instruction
—0—								n											
0	0	0	0	1	0	0	0	0	0	k					REPEAT #k				Repeat next instruction k+1 times
0	0	0	0	1	0	0	0	n							RCALL address				Push PC, PC += 2*s16
0	0	0	0	1	0	1	—								(Reserved)				
0	0	0	0	1	1	0	a	n							BRA Oa, addr				If accumulator an overflowed, PC += 2*simm16
0	0	0	0	1	1	1	a	n							BRA Sa, addr				If accumulator a saturated, PC += 2*simm16

0	**opcode**			**w**	**B**	**q**	**d**	**p**	**s**	**ALU operations: Wd ← OP(Ww,Ws)**				
0	0	0	1	0	w	B	q	d	p	s	SUBR[.B] Ww,Ws,Wd	C	Z	N	Wd ← Ws − Ww (Wd ← Ws + ~Ww + 1)
0	0	0	1	1	w	B	q	d	p	s	SUBBR[.B] Ww,Ws,Wd	C	Z	N	Wd ← Ws + ~Ww + C (Wd ← Ws − Ww − C̄)
0	0	1	0	k						d	MOV #k,Wd				Wd ← #imm16

0	**0**	**1**	**1**	**cond**				**n**	**Conditional branch to PC+2*n**				
0	0	1	1	0	0	0	0	n	BRA OV,addr				Branch if PSR.V is set
0	0	1	1	0	0	0	1	n	BRA C,addr				Branch if PSR.C is set
0	0	1	1	0	0	1	0	n	BRA Z,addr				Branch if PSR.Z is set
0	0	1	1	0	0	1	1	n	BRA N,addr				Branch if PSR.N is set
0	0	1	1	0	1	0	0	n	BRA LE,addr				Branch if PSR.Z, or PSR.N ≠ PSR.V
0	0	1	1	0	1	0	1	n	BRA LT,addr				Branch if PSR.N ≠ PSR.V
0	0	1	1	0	1	1	0	n	BRA LEU,addr				Branch if PSR.Z is set, or PSR.C is clear
0	0	1	1	0	1	1	1	n	BRA addr				Branch unconditionally
0	0	1	1	1	0	0	0	n	BRA NOV,addr				Branch if PSR.V is clear
0	0	1	1	1	0	0	1	n	BRA NC,addr				Branch if PSR.C is clear
0	0	1	1	1	0	1	0	n	BRA NZ,addr				Branch if PSR.Z is clear
0	0	1	1	1	0	1	1	n	BRA NN,addr				Branch if PSR.N is clear
0	0	1	1	1	1	0	0	n	BRA GT,addr				Branch if PSR.Z is clear, and PSR.N = PSR.V
0	0	1	1	1	1	0	1	n	BRA GE,addr				Branch if PSR.N = PSR.V
0	0	1	1	1	1	1	0	n	BRA GTU,addr				Branch if PSR.Z is clear, and PSR.C is set
0	0	1	1	1	1	1	1	n	(Reserved)				

0	**opcode**			**w**	**B**	**q**	**d**	**p**	**s**	**ALU operations: Wd ← OP(Ww,Ws)**				
0	1	0	0	0	w	B	q	d	p	s	ADD[.B] Ww,Ws,Wd	C	Z	N	Wd ← Ww + Ws

0	1	0	0	1	w		B	q	d		p	s	ADDC[.B] Ww,Ws,Wd	C	Z	N	Wd ← Ww + Ws + C
0	1	0	1	0	w		B	q	d		p	s	SUB[.B] Ww,Ws,Wd	C	Z	N	Wd ← Ww − Ws
0	1	0	1	1	w		B	q	d		p	s	SUBB[.B] Ww,Ws,Wd	C	Z	N	Wd ← Ww + ~Ws + C (Wd ← Ww − ~Ws − C̄)
0	1	1	0	0	w		B	q	d		p	s	AND[.B] Ww,Ws,Wd		Z	N	Wd ← Ww & Ws, logical and
0	1	1	0	1	w		B	q	d		p	s	XOR[.B] Ww,Ws,Wd		Z	N	Wd ← Ww ^ Ws, exclusive or
0	1	1	1	0	w		B	q	d		p	s	IOR[.B] Ww,Ws,Wd		Z	N	Wd ← Ww | Ws, inclusive or
0	1	1	1	1	w		B	q	d		p	s	MOV[.B] Ws,Wd		Z	N	Wd ← Ws (offset mode allowed)
1	0	0	0	0	f							d	MOV f,Wd				Wd ← f
1	0	0	0	1	f							s	MOV Ws,f				f ← Ws
1	0	0	1	0	k		B	k	d		k	s	MOV [Ws+s10],Wd				Load with 10-bit signed offset
1	0	0	1	1	k		B	k	d		k	s	MOV Ws,[Wd+s10]				Store with 10-bit signed offset
1	**0**	**1**	**0**	**0**	**opc**		**b**		**Z**	**B**	**0**	**0**	**0**	**p**	**s**	**Bit operations on Ws**	
1	0	1	0	0	0	0	0	b	0	B	0	0	0	p	s	BSET[.B] #b,Ws	Set bit b of Ws
1	0	1	0	0	0	0	1	b	0	B	0	0	0	p	s	BCLR[.B] #b,Ws	Clear bit b of Ws
1	0	1	0	0	0	1	0	b	0	B	0	0	0	p	s	BTG[.B] #b,Ws	Toggle bit b of Ws
1	0	1	0	0	0	1	1	b	0	0	0	0	0	p	s	BTST.C #b,Ws	Set PSR.C = bit b of Ws
1	0	1	0	0	0	1	1	b	1	0	0	0	0	p	s	BTST.Z #b,Ws	Set PSR.Z to complement of bit b of Ws
1	0	1	0	0	1	0	0	b	Z	0	0	0	0	p	s	BTSTS.z #b,Ws	Bit test (into C or Z), then set
1	0	1	0	0	1	0	1	Z	w		0	0	0	0	p	s	BTST.z Ww,Ws	Test bit (dynamic)
1	0	1	0	0	1	1	0	b		0	0	0	0	0	p	s	BTSS #b,Ws	Test bit b of Ws, skip if set
1	0	1	0	0	1	1	1	b		0	0	0	0	0	p	s	BTSS #b,Ws	Test bit b of Ws, skip if clear
1	**0**	**1**	**0**	**1**	**opc**		**b**	**f**								**Bit operations on f**	
1	0	1	0	1	0	0	0	b	f						b	BSET[.B] f, #b	Set bit b of f
1	0	1	0	1	0	0	1	b	f							BCLR.B #b,f	Clear bit b of f
1	0	1	0	1	0	1	0	b	f							BTG.B #b,f	Toggle bit b of f
1	0	1	0	1	0	1	1	b	f							BTST.B #b,f	Test bit b of f
1	0	1	0	1	1	0	0	b	f							BTSTS.B #b,f	Test bit b of f, then set
1	0	1	0	1	1	0	1	Z	w		0	0	0	0	p	s	BSW.z Ws,Ww	Copy PSW bit to bit Ww of Wb
1	0	1	0	1	1	1	0	b	f							BTSS #b,f	Test bit b of f, skip if set
1	0	1	0	1	1	1	1	b	f							BTSC #b,f	Test bit b of f, skip if clear
1	**0**	**1**	**1**	**0**	**0**	**opc**		**B**	**k**					**d**		**Register-immediate operations**	
1	0	1	1	0	0	0	0	0	B	k				d		ADD[.B] #u10,Wd	C N Z Wd ← Wd + imm10
1	0	1	1	0	0	0	0	1	B	k				d		ADC[.B] #u10,Wd	C N Z Wd ← Wd + imm10 + C
1	0	1	1	0	0	0	1	0	B	k				d		SUB[.B] #u10,Wd	C N Z Wd ← Wd − imm10
1	0	1	1	0	0	0	1	1	B	k				d		SUBB[.B] #u10,Wd	C N Z Wd ← Wd − imm10 − C̄
1	0	1	1	0	0	1	0	0	B	k				d		AND[.B] #u10,Wd	N Z Wd ← Wd & imm10

1	0	1	1	0	0	1	0	1	B	k	d	XOR[.B] #u10,Wd		N	Z	Wd ← Wd ^ imm10
1	0	1	1	0	0	1	1	0	B	k	d	IOR[.B] #u10,Wd		N	Z	Wd ← Wd | imm10
1	0	1	1	0	0	1	1	1	B	k	d	MOV[.B] #u10,Wd				Wd ← imm10

1	**0**	**1**	**1**	**0**	**1**	**opc**			**B**	**D**	**f**	**ALU operations: dest ← OP(f,W0)**				
1	0	1	1	0	1	0	0	0	B	D	f	ADD[.B] f[,WREG]	C	N	Z	dest ← f + W0
1	0	1	1	0	1	0	0	1	B	D	f	ADC[.B] f[,WREG]	C	N	Z	dest ← f + W0 + C
1	0	1	1	0	1	0	1	0	B	D	f	SUB[.B] f[,WREG]	C	N	Z	dest ← f – W0
1	0	1	1	0	1	0	1	1	B	D	f	SUBB[.B] f[,WREG]	C	N	Z	dest ← f – W0 + C̄
1	0	1	1	0	1	1	0	0	B	D	f	AND[.B] f[,WREG]		N	Z	dest ← f & W0
1	0	1	1	0	1	1	0	1	B	D	f	XOR[.B] f[,WREG]		N	Z	dest ← f ^ W0
1	0	1	1	0	1	1	1	0	B	D	f	IOR[.B] f[,WREG]		N	Z	dest ← f | W0
1	0	1	1	0	1	1	1	1	B	1	f	MOV[.B] WREG,f				dest ← W0

1	**0**	**1**	**1**	**1**	**0**	**0**	**op**		**w**	**d**	**0**	**p**	**s**	**16×16→32 multiplication**		
1	0	1	1	1	0	0	0	0	w	d	0	p	s	MUL.UU Ww,Ws,Wd				Wd+1:Wd ← Wb * Ws (unsigned)
1	0	1	1	1	0	0	0	1	w	d	0	p	s	MUL.US Ww,Ws,Wd				Wd+1:Wd ← Wb * Ws (Ws signed)
1	0	1	1	1	0	0	1	0	w	d	0	p	s	MUL.SU Ww,Ws,Wd				Wd+1:Wd ← Wb * Ws (Wb signed)
1	0	1	1	1	0	0	1	1	w	d	0	p	s	MUL.SS Ww,Ws,Wd				Wd+1:Wd ← Wb * Ws (signed)

1	**0**	**1**	**1**	**1**	**0**	**1**	**op**		**w**		**d**	**p**	**s**	**Program memory access**		
1	0	1	1	1	0	1	0	0	B	qqq	d	p	s	TBLRDL [Ws],Wd				Wd ← PROGRAM[TBLPAG:Ws] (bits 15:0)
1	0	1	1	1	0	1	0	1	B	qqq	d	p	s	TBLRDH [Ws],Wd				Wd ← PROGRAM[TBLPAG:Ws] (bit 23:16)
1	0	1	1	1	0	1	1	0	B	qqq	d	p	s	TBLWTL Ws,[Wd]				PROGRAM[TBLPAG:Wd] ← Ws (bits 15:0)
1	0	1	1	1	0	1	1	1	B	qqq	d	p	s	TBLWTH [Ws],[Wd]				PROGRAM[TBLPAG:Wd] ← Ws (bit 23:16)

1	0	1	1	1	1	0	0	0	B	0	f				MUL[.B] f				W3:W2 ← f * W0 (unsigned)
1	0	1	1	1	1	0	1	—							(Reserved)				
1	0	1	1	1	1	1	0	0	0	000	d	0	p	s	MOV.D Ws,Wd				Move register pair (source may be memory)
1	0	1	1	1	1	1	0	1	0	q	d	000	s	0	MOV.D Ws,Wd				Move register pair (dest may be memory)
1	0	1	1	1	1	1	1	—							(Reserved)				

1	**1**	**0**	**0**	**0**	**m**		**A**	**S**	**x**	**y**	**i**	**j**	**a**	**DSP MAC (dsPIC only)**		
1	**1**	**0**	**0**	**1**										**Other DSP instructions (dsPIC only)**		
1	1	0	0	1	1	1	1	0	0	000	d	p	s	FF1R Ws,Wd				Find first one from right (lsb)

1	1	0	0	1	1	1	1	1	0	000	d	p	s	FF1L Ws,Wd				Find first one from left (msb)
1	**1**	**0**	**1**	**0**	**0**	**opc**			**B**	**q**	**d**	**p**	**s**	**Shift/rotate W register**				
1	1	0	1	0	0	0	0	0	B	q	d	p	s	SL[.B] Ws,Wd	C	N	Z	Wd ← Ws << 1, shift left (into carry)
1	1	0	1	0	0	0	1	0	B	q	d	p	s	LSR[.B] Ws,Wd	C	N	Z	Wd ← Ws >> 1, logical shift right
1	1	0	1	0	0	0	1	1	B	q	d	p	s	ASR[.B] Ws,Wd	C	N	Z	Wd ← Ws >> 1, arithmetic shift right
1	1	0	1	0	0	1	0	0	B	q	d	p	s	RLNC[.B] Ws,Wd		N	Z	Wd ← Ws <<< 1, rotate left (no carry)
1	1	0	1	0	0	1	0	1	B	q	d	p	s	RLC[.B] Ws,Wd	C	N	Z	C:Wd ← Ws:C << 1, rotate left through carry
1	1	0	1	0	0	1	1	0	B	q	d	p	s	RRNC[.B] Ws,Wd		N	Z	Wd ← Ws >>> 1, rotate right (no carry)
1	1	0	1	0	0	1	1	1	B	q	d	p	s	RRC[.B] Ws,Wd	C	N	Z	Wd:C ← C:Ws >> 1, rotate right through carry
1	**1**	**0**	**1**	**0**	**1**	**opc**			**B**	**D**	**f**			**Shift/rotate f**				
1	1	0	1	0	1	0	0	0	B	D	f			SL[.B] f[,WREG]	C	N	Z	dest ← f << 1, shift left (into carry)
1	1	0	1	0	1	0	1	0	B	D	f			LSR[.B] f[,WREG]	C	N	Z	dest ← f >> 1, logical shift right
1	1	0	1	0	1	0	1	1	B	D	f			ASR[.B] f[,WREG]	C	N	Z	dest ← f >> 1, arithmetic shift right
1	1	0	1	0	1	1	0	0	B	D	f			RLNC[.B] f[,WREG]		N	Z	dest ← f <<< 1, rotate left (no carry)
1	1	0	1	0	1	1	0	1	B	D	f			RLC[.B] f[,WREG]	C	N	Z	C:dest ← f:C << 1, rotate left through carry
1	1	0	1	0	1	1	1	0	B	D	f			RRNC[.B] f[,WREG]		N	Z	dest ← f >>> 1, rotate right (no carry)
1	1	0	1	0	1	1	1	1	B	D	f			RRC[.B] f[,WREG]	C	N	Z	dest:C ← C:f >> 1, rotate right through carry
1	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**U**	**t**		**v**	**W**	**0**	**0**	**s**	**32/16 and 16/16 divide steps (prefix with REPEAT #17)**				
1	1	0	1	1	0	0	0	0	0000		v	0	0	0	s	DIV.S Wv,Ws	C	N	Z	Divide step, W0 ← Wv/Ws, W1 ← remainder
1	1	0	1	1	0	0	0	0	t		v	1	0	0	s	DIV.SD Wv,Ws	C	N	Z	Divide step, W0 ← Wt:Wv/Ws, W1 ← remainder
1	1	0	1	1	0	0	0	1	0000		v	0	0	0	s	DIV.U Wv,Ws	C	N	Z	Divide step, W0 ← Wv/Ws, W1 ← remainder
1	1	0	1	1	0	0	0	1	t		v	1	0	0	s	DIV.UD Wv,Ws	C	N	Z	Divide step, W0 ← Wt:Wv/Ws, W1 ← remainder
1	1	0	1	1	0	0	1	0	t		0 0 0 0	0	0	0	s	DIVF Wt,Ws	C	N	Z	Divide step, W0 ← Wt:0/Ws, W1 ← remainder
1	1	0	1	1	0	1	—									(Reserved)				
1	1	0	1	1	1	0	0	—								(Reserved)				
1	1	0	1	1	1	0	0	0	w		d	0	0	0	s	SL Ww,Ws,Wd		N	Z	Wd ← Wv << Ws

1	1	0	1	1	1	0	0	0	w			d	1	0	0	k	SL Ww,#u4,Wd		N	Z	Wd ← Wv << imm4
1	1	0	1	1	1	1	0	0	w			d	0	0	0	s	LSR Ww,Ws,Wd		N	Z	Wd ← Wv>> Ws, logical shift right
1	1	0	1	1	1	1	0	0	w			d	1	0	0	k	LSR Ww,#u4,Wd		N	Z	Wd ← Wv>> imm4, logical shift right
1	1	0	1	1	1	1	0	1	w			d	0	0	0	s	ASR Ww,Ws,Wd		N	Z	Wd ← Wv>> Ws, arithmetic shift right
1	1	0	1	1	1	1	0	1	w			d	1	0	0	k	ASR Wv,#u4,Wd		N	Z	Wd ← Wv>> imm4, arithmetic shift right
1	1	0	1	1	1	1	1	0	0000			d	p			s	FBCL Ws,Wd	C			Find permissible arithmetic normalization shift
1	1	1	0	0	0	0	0	0	0000			B	0	0	0	p	s	CP0[.B] Ws	C	N	Z	Compare with zero, Ws – 0
1	1	1	0	0	0	0	1	0	w			B	0	0	0	p	s	CP[.B] Ww,Ws	C	N	Z	Compare, Wb – Ws (Wb + ~Ws + 1)
1	1	1	0	0	0	0	1	1	w			B	0	0	0	p	s	CPB[.B] Ww,Ws	C	N	Z	Compare with borrow, Wb + ~Ws + C (Wb – Ws – C̄)
1	1	1	0	0	0	1	0	0	B	0	f							CP0[.B] Ws	C	N	Z	Compare with zero, f – 0
1	1	1	0	0	0	1	1	0	B	0	f							CP[.B] f	C	N	Z	Compare f, f – W0
1	1	1	0	0	0	1	1	1	B	0	f							CPB[.B] f	C	N	Z	Compare f with borrow, f + ~W0 + C (f – W0 – C̄)
1	1	1	0	0	1	0	—											(Reserved)				
1	1	1	0	0	1	1	0	0	w			B	0	0	0	0	0	0	s	CPSGT[.B] Ww,Ws				Compare and skip if greater than (Wb > Ws, signed)
1	1	1	0	0	1	1	0	1	w			B	0	0	0	0	0	0	s	CPSLT[.B] Ww,Ws				Compare and skip if less than (Wb < Ws, signed)
1	1	1	0	0	1	1	1	0	w			B	0	0	0	0	0	0	s	CPSNE[.B] Ww,Ws				Compare and skip if not equal (Wb ≠ Ws)
1	1	1	0	0	1	1	1	1	w			B	0	0	0	0	0	0	s	CPSNE[.B] Ww,Ws				Compare and skip if equal (Wb = Ws)
1	1	1	0	1	0	0	0	0	B	q		d	p			s		INC[.B] Ws,Wd	C	N	Z	Wd ← Ws+1
1	1	1	0	1	0	0	0	1	B	q		d	p			s		INC2[.B] Ws,Wd	C	N	Z	Wd ← Ws+2
1	1	1	0	1	0	0	1	0	B	q		d	p			s		DEC[.B] Ws,Wd	C	N	Z	Wd ← Ws−1
1	1	1	0	1	0	0	1	1	B	q		d	p			s		DEC2[.B] Ws,Wd	C	N	Z	Wd ← Ws−2
1	1	1	0	1	0	1	0	0	B	q		d	p			s		NEG[.B] Ws,Wd	C	N	Z	Wd ← ~Ws+1
1	1	1	0	1	0	1	0	1	B	q		d	p			s		COM[.B] Ws,Wd		N	Z	Wd ← ~Ws
1	1	1	0	1	0	1	1	0	B	q		d	000			0000		CLR[.B] Wd				Wd ← 0
1	1	1	0	1	0	1	1	1	B	q		d	000			0000		SETM[.B] Wd				Wd ← ~0
1	1	1	0	1	1	0	0	0	B	D	f							INC[.B] f[,WREG]	C	N	Z	dest ← f+1
1	1	1	0	1	1	0	0	1	B	D	f							INC2[.B] f[,WREG]	C	N	Z	dest ← f+2
1	1	1	0	1	1	0	1	0	B	D	f							DEC[.B] f[,WREG]	C	N	Z	dest ← f−1
1	1	1	0	1	1	0	1	1	B	D	f							DEC[.B] f[,WREG]	C	N	Z	dest ← f−2
1	1	1	0	1	1	1	0	0	B	D	f							NEG[.B] f[,WREG]	C	N	Z	dest ← ~f+1
1	1	1	0	1	1	1	0	1	B	D	f							COM[.B] f[,WREG]		N	Z	dest ← ~f

1	1	1	0	1	1	1	1	0	B	D	f				CLR[.B] f[,WREG]				dest ← 0
1	1	1	0	1	1	1	1	1	B	D	f				SETM[.B] f[,WREG]				dest ← ~0
1	**1**	**1**	**1**	**0**	**0**	**m**	**A**	**1**	**x**	**y**	**i**	**j**	**opc**		**DSP MPY/MAC/ED/EDAC (dsPIC only)**				
1	1	1	1	0	1	—									(Reserved)				
1	1	1	1	1	0	0	0	f						0	PUSH f				Push f on top of stack
1	1	1	1	1	0	0	1	f						0	POP f				Pop f from top of stack
1	1	1	1	1	0	1	0	0	0	k					LNK #u14				Push W14, W14 ← W15, W15 += u14
1	1	1	1	1	0	1	0	1	0	—0—					ULNK				W15 ← W14, pop W14
1	1	1	1	1	0	1	1	0	0	000	d	p	s		SE Ws,Wd	C	N	Z	Wd ← sign_extend(Ws), copy bit 7 to bits 15:8
1	1	1	1	1	0	1	1	1	0	000	d	p	s		ZE Ws,Wd	1	0	Z	Wd ← zero_extend(Ws), clear bits 15:8
1	1	1	1	1	1	0	0	0	0	k					DISI #u14				Disable interrupt for k+1 cycles
1	1	1	1	1	1	0	1	0	0	000	d	000	s		EXCH Ws,Wd				Swap contents of registers Ws, Wd
1	1	1	1	1	1	0	1	0	1	000	0000	000	s		DAW.B Ws	C			Decimal adjust based on C, DC
1	1	1	1	1	1	0	1	1	B	000	0000	000	s		SWAP[.B] Ws				Swap halves of Ws
1	1	1	1	1	1	1	0	0	0	—0—					RESET				Software reset
1	1	1	1	1	1	1	0	0	1	0	—0—			k	PWRSAV #u1				Go into sleep or idle mode
1	1	1	1	1	1	1	0	0	1	1	—0—				CLRWDT				Clear watchdog timer
1	1	1	1	1	1	1	0	1	0	0	—0—				POP.S				Pop shadow registers (W0–3, part of PSR)
1	1	1	1	1	1	1	0	1	0	1	—0—				PUSH.S				Push shadow registers (W0–3, part of PSR)
1	1	1	1	1	1	1	0	1	1	—					(Reserved)				
1	1	1	1	1	1	1	1	1	—						NOPR				No operation (version #2)

PIC32 32-bit microcontrollers

In November 2007 Microchip introduced the new PIC32MX [18] family of 32-bit microcontrollers. The initial device line-up is based on the industry standard MIPS32 M4K Core.[19] The device can be programmed using the Microchip MPLAB C Compiler for PIC32 MCUs [20], a variant of the GCC compiler. The first 18 models currently in production (PIC32MX3xx and PIC32MX4xx) are pin to pin compatible and share the same peripherals set with the PIC24FxxGA0xx family of (16-bit) devices allowing the use of common libraries, software and hardware tools. Today starting at 28 pin in small QFN packages up to high performance devices with Ethernet, CAN and USB OTG, full family range of mid-range 32-bit microcontrollers are available.

The PIC32 architecture brings a number of new features to Microchip portfolio, including:

- The highest execution speed 80 MIPS (120+[21] Dhrystone MIPS @ 80 MHz)
- The largest flash memory: 512 kByte
- One instruction per clock cycle execution
- The first cached processor
- Allows execution from RAM

- Full Speed Host/Dual Role and OTG USB capabilities
- Full JTAG and 2 wire programming and debugging
- Real-time trace

Device variants and hardware features

PIC devices generally feature:

- Sleep mode (power savings).
- Watchdog timer.
- Various crystal or RC oscillator configurations, or an external clock.

Variants

Within a series, there are still many device variants depending on what hardware resources the chip features.

- General purpose I/O pins.
- Internal clock oscillators.
- 8/16/32 Bit Timers.
- Internal EEPROM Memory.
- Synchronous/Asynchronous Serial Interface USART.
- MSSP Peripheral for I²C and SPI Communications.
- Capture/Compare and PWM modules.
- Analog-to-digital converters (up to ~1.0 MHz).
- USB, Ethernet, CAN interfacing support.
- External memory interface.
- Integrated analog RF front ends (PIC16F639, and rfPIC).
- KEELOQ Rolling code encryption peripheral (encode/decode)
- And many more.

Trends

The first generation of PICs with EPROM storage are almost completely replaced by chips with Flash memory. Likewise, the original 12-bit instruction set of the PIC1650 and its direct descendants has been superseded by 14-bit and 16-bit instruction sets. Microchip still sells OTP (one-time-programmable) and windowed (UV-erasable) versions of some of its EPROM based PICs for legacy support or volume orders. The Microchip website lists PICs that are not electrically erasable as OTP. UV erasable windowed versions of these chips can be ordered.

Part number suffixes

The F in a name generally indicates the PICmicro uses flash memory and can be erased electronically. Conversely, a C generally means it can only be erased by exposing the die to ultraviolet light (which is only possible if a windowed package style is used). An exception to this rule is the PIC16C84 which uses EEPROM and is therefore electrically erasable.

An L in the name indicates the part will run at a lower voltage, often with frequency limits imposed.[]

Parts designed specifically for low voltage operation, within a strict range of 3 - 3.6 volts, are marked with a J in the part number. These parts are also uniquely I/O tolerant as they will accept up to 5 V as inputs.[]

PIC clones

Third party manufacturers make compatible products, for example the Parallax SX.

Development tools

Microchip provides a freeware IDE package called MPLAB, which includes an assembler, linker, software simulator, and debugger. They also sell C compilers for the PIC18 and dsPIC which integrate cleanly with MPLAB. Free student versions of the C compilers are also available with all features. But for the free versions, optimizations will be disabled after 60 days.[22]

Several third parties make C language compilers for PICs, many of which integrate to MPLAB and/or feature their own IDE. A fully featured compiler for the PICBASIC language to program PIC microcontrollers is available from meLabs, Inc. Mikroelektronika offers PIC compilers in C, Basic and Pascal programming languages.

A graphical programming language, Flowcode, exists capable of programming 8 and 16 bit PIC devices and generating PIC compatible C code. It exists in numerous versions from a free demonstration to a more complete professional edition.

The only opensource compiler for the PIC16 and PIC18 family is the SDCC. It make use of GPutils for linker and assembler tools. A plugin is needed to install them in MPLAB or MPLABX.[23]

Development tools are available for the PIC family under the GPL or other free software or open source licenses.

Device programmers

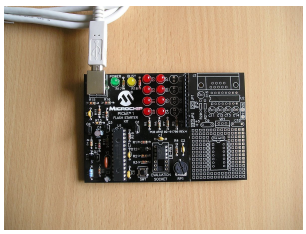
Devices called "programmers" are traditionally used to get program code into the target PIC. Most PICs that Microchip currently sell feature ICSP (In Circuit Serial Programming) and/or LVP (Low Voltage Programming) capabilities, allowing the PIC to be programmed while it is sitting in the target circuit. ICSP programming is performed using two pins, clock and data, while a high voltage (12V) is present on the Vpp/MCLR pin. Low voltage programming dispenses with the high voltage, but reserves exclusive use of an I/O pin and can therefore be disabled to recover the pin for other uses (once disabled it can only be re-enabled using high voltage programming).

There are many programmers for PIC microcontrollers, ranging from the extremely simple designs which rely on ICSP to allow direct download of code from a host computer, to intelligent programmers that can verify the device at several supply voltages. Many of these complex programmers use a pre-programmed PIC themselves to send the programming commands to the PIC that is to be programmed. The intelligent type of programmer is needed to program earlier PIC models (mostly EPROM type) which do not support in-circuit programming.

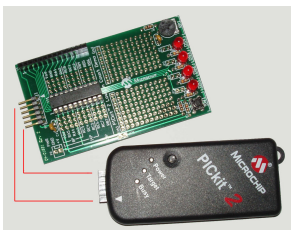
Many of the higher end flash based PICs can also self-program (write to their own program memory). Demo boards are available with a small bootloader factory programmed that can be used to load user programs over an interface such as RS-232 or USB, thus obviating the need for a programmer device. Alternatively there is bootloader firmware available that the user can load onto the PIC using ICSP. The advantages of a bootloader over ICSP is the far superior programming speeds, immediate program execution following programming, and the ability to both debug and program using the same cable.

Programmers/debuggers are available directly from Microchip. Third party programmers range from plans to build your own, to self-assembly kits and fully tested ready-to-go units. Some are simple designs which require a PC to do the low-level programming signalling (these typically connect to the serial or parallel port and consist of a few simple components), while others have the programming logic built into them (these typically use a serial or USB connection, are usually faster, and are often built using PICs themselves for control).

The following are the official PICkit programmer/debuggers from Microchip:



Microchip PICkit1



Microchip PICkit2

Microchip PICkit3

PICKit 2 clones and open source

PICKit 2 has been an interesting PIC programmer from Microchip. It can program all PICs and debug most of the PICs (as of May-2009, only the PIC32 family is not supported for MPLAB debugging). Ever since its first releases, all software source code (firmware, PC application) and hardware schematic are open to the public. This makes it relatively easy for an end user to modify the programmer for use with a non-Windows operating system such as Linux or Mac OS. In the mean time, it also creates lots of DIY interest and clones. This open source structure brings many features to the PICKit 2 community such as Programmer-to-Go, the UART Tool and the Logic Tool, which have been contributed by PICKit 2 users. Users have also added such features to the PICKit 2 as 4MB Programmer-to-go capability, USB buck/boost circuits, RJ12 type connectors and others.

Debugging

Software emulation

Commercial and free emulators exist for the PIC family processors.

In-circuit debugging

Later model PICs feature an ICD (in-circuit debugging) interface, built into the CPU core. ICD debuggers (MPLAB ICD2 and other third party) can communicate with this interface using three lines. This cheap and simple debugging system comes at a price however, namely limited breakpoint count (1 on older pics 3 on newer PICs), loss of some IO (with the exception of some surface mount 44-pin PICs which have dedicated lines for debugging) and loss of some features of the chip. For small PICs, where the loss of IO caused by this method would be unacceptable, special headers are made which are fitted with PICs that have extra pins specifically for debugging.

In-circuit emulators

Microchip offers three full in-circuit emulators: the MPLAB ICE2000 (parallel interface, a USB converter is available); the newer MPLAB ICE4000 (USB 2.0 connection); and most recently, the REAL ICE. All of these ICE tools can be used with the MPLAB IDE for full source-level debugging of code running on the target.

The ICE2000 requires emulator modules, and the test hardware must provide a socket which can take either an emulator module, or a production device.

The REAL ICE connects directly to production devices which support in-circuit emulation through the PGC/PGD programming interface, or through a high speed connection which uses two more pins. According to Microchip, it supports "most" flash-based PIC, PIC24, and dsPIC processors.[24]

The ICE4000 is no longer directly advertised on Microchip's website, and the purchasing page states that it is not recommended for new designs.

References

[1] http://ww1.microchip.com/downloads/en/DeviceDoc/39630C.pdf
[2] http://www.datasheetarchive.com/dl/Databooks-1/Book241-407.pdf
[3] "PICmicro Family Tree", PIC16F Seminar Presentation http://www.microchip.com.tw/PDF/2004_spring/PIC16F%20seminar%20presentation.pdf
[4] "MOS DATA 1976", General Instrument 1976 Databook
[5] "1977 Data Catalog", Micro Electronics from General Instrument Corporation http://www.rhoent.com/pic16xx.pdf
[6] http://ww1.microchip.com/downloads/en/DeviceDoc/35007b.pdf
[8] Microchip Product Selector (http://www.microchip.com/productselector/MCUProductSelector.html)
[9] "PIC Paging and PCLATH" (http://massmind.org/techref/microchip/pages.htm)
[11] http://www.emc.com.tw/eng/products.asp
[13] http://mdubuc.freeshell.org/Sdcc/
[14] http://www.microchipc.com/sourcecode/
[16] (http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2018&mcparam=en013529)
[18] http://ww1.microchip.com/downloads/en/DeviceDoc/61177a.pdf
[19] http://www.mips.com/products/processors/32-64-bit-cores/mips32-m4k/
[20] http://microchip.com/c32

External links

- PIC microcontroller (http://www.dmoz.org/Computers/Hardware/Components/Processors/PIC//) at the Open Directory Project.
- Official Microchip website (http://www.microchip.com/)
- PIC wifi projects website (http://www.libstock.com/project_categories/view/25/wifi/)

Article Sources and Contributors

PIC microcontroller *Source*: http://en.wikipedia.org/w/index.php?oldid=570483753 *Contributors*: .digamma, 10metreh, 28421u2232nfenfcenc, 2A00:1398:4:0:6E62:6DFF:FE5F:71E, A little insignificant, A. B., Abdull, Adam1213, Adambro, Ahoerstemeier, Akilaa, Alansohn, Alecv, Aleksandarbrain, AlexOvShaolin, Alexdzm90, Alistair1978, Allens, Allstarecho, Alvin-cs, Andy Dingley, Angelpeream, ArglebargleIV, Atomsmith, Attilios, Avochelm, Azhyd, Barefoottech, Barek, Bejnar, Bernard Teo, Betatester228, Bgwhite, Bmearns, Bo102010, Bobo192, Bombadier337, Bradediger, Brammers, BrianWilloughby, Brighterorange, BrotherE, Brouhaha, Browsem, Bsodmike, Bugfarmer, CSumit, Calltech, CanOfWorms, CanisRufus, Carveone, Cashimor, Choppingmall, Chowbok, Chris Roy, Chris the speller, ChrisJ, Ckielstra, Colonies Chris, Cometstyles, CommonsDelinker, Cyanoir, CyrilB, Dan Gardner, Danara5, Danim, DavesPlanet, DavidCary, Davitenio, Dcamp314, Devon Sean McCullough, Dhx1, DiamondDevil, Dicklyon, DimychUA, Dirac1933, Dirkbb, Dogcow, Ds13, Electron9, Eplondke, Erpingham, EvanCarroll, Eworldprojects, Excirial, Fabero74, Ferritecore, Fiducial, Firsfron, Fixentries, FourBlades, Fransschreuder, Frap, Frecklefoot, Furrykef, Gadlen, Gaius Cornelius, Gauravsangwan, Gcm, Gene Nygaard, Gengiskanhg, GermanX, Gfutia, Gilliam, Glenn, Goldzen, GraemeL, Greglecuyer, Guy Macon, Gwernol, Hawk777, Hellisp, HenkeB, Heracles31, Heron, Hithisishal, Hyvatti, Ihope127, Ikmac, Imroy, Insanity Incarnate, Intersofia, Izx, Jaanus.kalde, Jahoe, James086, Jamodio, Japo, Jason One, Javawizard, Jburstein, Jd4x4, Jerryobject, JesseHogan, Jfraser, Jhallenworld, Jianhui67, Johnpeterharvey, Jon-ecm, JonHarder, Jonathanwagner, Jonny wdrw, Jrash, Jwortzel, Karlwk, Katalaveno, Kateshortforbob, Kevin E Hawkins, Kinema, Kizor, KnowledgeOfSelf, Knuckx, Kubing, Kuru, Kyxui, Lauri.pirttiaho, Lerdthenerd, Liqk, Lmblackjack21, Lraingele, Lrb13615, MZMcBride, Ma3nocum, Mac, Magioladitis, Mahjongg, Margin1522, Mat-C, Mateo LeFou, Materialscientist, Matt B., MatthewJBennett, Matthewmadmad, Maury Markowitz, Mboverload, Mdenton, Megamix, Miceduan, Michael Hardy, MichaelFrey, Mikemurphy, Mintleaf, MohammadEbrahim, Monkeyman, Morcheeba, Mortense, MortimerCat, Morwen, Moxfyre, Mr Stephen, Mr z, Nabokov, Nat1192, Niggurath, NightFalcon90909, Nishkid64, NobbiP, Nuno Tavares, Omegatron, Onorai, OpBanana, Outback the koala, PJohnson, PPA94, PSL, Pazza pazza, Peter todd, Philip Trueman, Pip2andahalf, Plaasjaapie, Plasticup, Plugwash, Ppapadeas, RCX, Ramsey585, Randomblue, RedWolf, Redfarmer, Reelrt, RenamedUser01302013, Rich Farmbrough, Rjwilmsi, Rod57, Ronz, S.riccardelli, SJP, Saimhe, Samwb123, Satellizer, Sbmeirow, Sbogdanov, Shadowjams, SheepNotGoats, Shjacks45, Shoez, Skeeter2, Smishek, Smt52, Softy, Speedevil, SpuriousQ, Stan Shebs, Stingraze, Stjrna, Strigoides, Sv1xv, Sxpilot250, Tabletop, Taemyr, Talkie tim, TechPurism, Tempodivalse, Thunderchild, Tide rolls, Toddsoc, Tom Morris, Toussaint, Trialsanderrors, Turbo852, Una Smith, Uzume, Val42, Vary, Velociostrich, Vic93, Voidxor, WardXmodem, Warnockm, Washburnmav, Wavelength, Wdfarmer, Weopgon, Wernher, Wik, Wiki alf, Wikiwooroo, WimHeirman, Wjw1961, Wtshymanski, X201, Yunshui, ZaferXYZ, Zetawoof, Zoicon5, Zzuuzz, 806 anonymous edits

Image Sources, Licenses and Contributors

Image:PIC microcontrollers.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:PIC_microcontrollers.jpg *License*: Public Domain *Contributors*: MikeMurphy

Image:Microchip PIC24HJ32GP202.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Microchip_PIC24HJ32GP202.jpg *License*: Creative Commons Attribution-Share Alike *Contributors*: User:Acdx, User:GeorgHH

Image:PIC12C508-HD.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:PIC12C508-HD.jpg *License*: Creative Commons Attribution 3.0 *Contributors*: Dhx1

Image:PIC16C505-HD.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:PIC16C505-HD.jpg *License*: Creative Commons Attribution 3.0 *Contributors*: Dhx1

Image:PIC16CxxxWIN.JPG *Source*: http://en.wikipedia.org/w/index.php?title=File:PIC16CxxxWIN.JPG *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: Camillo, Glenn

Image:Pickit1 top.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Pickit1_top.jpg *License*: Copyrighted free use *Contributors*: User:Dhenry

Image:PICkit2.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:PICkit2.jpg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: User:NobbiP

Image:PICKit3.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:PICKit3.jpg *License*: Copyrighted free use *Contributors*: Glossywhite

License