The picoProcessor ISA

Introduction

The picoProcessor (pP) is an 8-bit processor intended for educational purposes. It is similar to 8-bit microprocessors for small embedded applications, but has an instruction set architecture more similar to RISC processors.

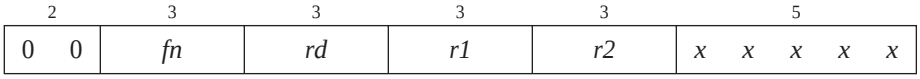
The pP has separate instruction and data memories. The instruction memory is 4K instructions in size, and the data memory is 256 bytes. The pP can also address I/O devices using up to 256 input ports and 256 output ports.

Within the processor there are eight 8-bit general purposes registers, r0 to r7. R0 is always read as zero and ignores writes. There is also a return-address stack of implementation-defined depth (at least four entries), an interrupt return register and Zero (Z) and Carry (C) condition codes.

Instruction Encoding

Instructions in the pP are all 19 bits long, and are encoded in several formats.

ALU Reg-Reg Instructions

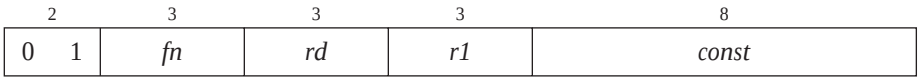


This format is used for arithmetic and logical instructions that operate on two register values (*r1* and *r2*). The result is stored in a destination register (*rd*). The function code (*fn*) values are:

Instruction	Operation	fn
ADD	Add without carry in	000
ADDC	Add with carry in	001
SUB	Subtract without carry in	010
SUBC	Subtract with carry in	011
AND	Bitwise logical and	100
OR	Bitwise logical inclusive or	101
XOR	Bitwise logical exclusive or	110
MSK	Bitwise logical mask (and-not)	111

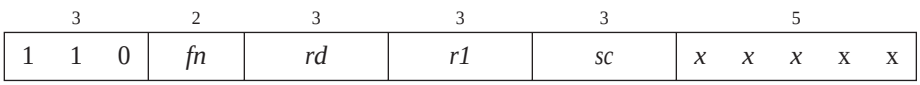
The Z condition code is set to 1 if the result is zero, otherwise it is set to 0. For `ADD` and `ADDC`, the C condition code is the carry out of the addition. For `SUB` and `SUBC`, C is the borrow out of the subtraction and indicates that the operation overflowed to a negative result. For logical operations, C is always set to 0.

ALU Reg-Immed Instructions



This format is used for arithmetic and logical instructions that operate on a register value (*r1*) and an immediate constant value (*const*). The result is stored in a destination register (*rd*). The function code (*fn*) values and the condition code settings are the same as for ALU Reg-Reg instructions.

Shift Instructions

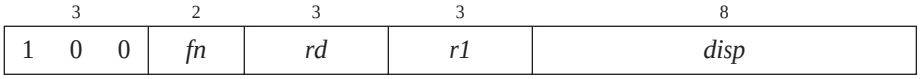


This format is used for shift and rotate instructions that operate on a register value (*r1*). The number of bit positions by which the value is shifted or rotated is given by the shift count (*sc*). The result is stored in a destination register (*rd*). The shift function code (*fn*) values are:

Instruction	Operation	fn
SHL	Shift left	00
SHR	Shift right	01
ROL	Rotate left	10
ROR	Rotate right	11

The Z condition code is set to 1 if the result is zero, otherwise it is set to 0. The C condition code is set to the value of the bit shifted or rotated out of the operand value.

Memory and I/O Instructions

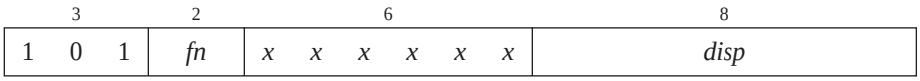


This format is used for memory and I/O load and store instructions. The effective address is calculated by adding a signed displacement (*disp*) to the value of a base address register (*r1*). For memory instructions, the effective address is used to access the data memory, and for I/O instructions, it is used as the I/O port number. For oad instructions, *rd* is the destination register, and for store instruction, *rd* is the source register. The function code (*fn*) values are:

Instruction	Operation	fn
LDM	Load from memory	00
STM	Store to memory	01
INP	Input from port	10
OUT	Output to port	11

The Z and C condition codes are unaffected by these instructions.

Conditional Branch Instructions

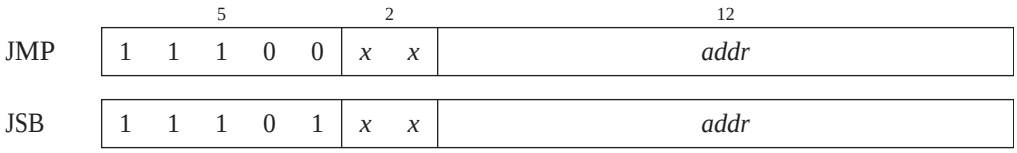


This format is used for conditional branch instructions. The target address is calculated by adding a signed displacement (*disp*) to the address of the instruction following the branch. If the condition specified by the branch is true, control is transferred to the instruction at the target address; otherwise control continues with the instruction following the branch. The function code (*fn*) values are:

Instruction	Operation	fn
BZ	Branch if zero	00
BNZ	Branch if not zero	01
BC	Branch if carry	10
BNC	Branch if not carry	11

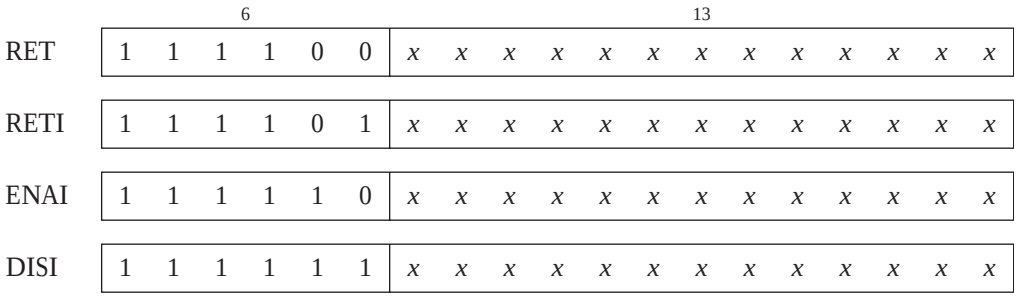
The Z and C condition codes are unaffected by these instructions.

Jump Instructions



This format is used for unconditional jump instructions. Control is transferred to the instruction at the target address (*addr*). The `JSB` instruction, however, first pushes the address of the instruction following the `JSB` onto the return address stack. For both instructions, the condition codes are unaffected.

Miscellaneous Instructions



This format is used for miscellaneous instructions that have no operands. `RET` returns from a subroutine by popping the top of the return address stack to the program counter. `RETI` restores the saved program counter and condition code values from the interrupt register. `ENAI` enables interrupts, and `DISI` disables interrupts. The `RET`, `ENAI` and `DISI` instructions do not affect the condition codes.

Interrupt and Reset

The pP has an input for requesting an interrupt. When interrupts are enabled and the interrupt request is active at the start of an instruction fetch sequence, the program counter and condition codes are copied to the interrupt register, interrupts are disabled, and control is transferred to program address 1. Interrupt service code should be located at that address. Note that, since there is only one interrupt register, nested interrupts are not supported.

The interrupt service code must return to the interrupted program by executing a `RETI` instruction, which restores the saved condition codes and program counter from the interrupt register and re-enables interrupts.

Upon system reset, the pP disables interrupts and starts executing instructions from address 0. In a system that uses interrupts, the interrupt service code starts at address 1. Thus, the instruction at address 0 would normally be a jump to initialization code.