The Phoenix (Revision 2) architecture

Overview

Phoenix is an 8-bit architecture with 16-bit memory and io addressing. The architecture has dedicated memory and io address spaces. Instructions and data are mixed.

Instruction encoding

An instruction has a fixed size of 24-bit. An instruction can contain 2 registers and an 8-bit intermediate value or a 16-bit intermediate value.

00000000 0000 0000 00000000

Instuction opcode

Second register

First register

8-bit Intermediate

To store a 16-bit intermediate value you need to combine the register section and the 8-bit intermediate section.

Halting execution

To halt execution, you must either jump to 0xffff (the end of the memory space) or set the HALT flag.

Registers

General purpose registers

There are 16 general purpose registers with a size of 8-bit. Those are named R0 – R15. Those registers can be used with all instructions accepting a register.

Index registers

There are 2 index registers with a size of 16 bits. These are called A and B. An index register consists of 2 general purpose registers. These registers are encoded in the instruction opcode instead of the register part.

- Index register A consists of r0 (low byte) and r1 (high byte).
- Index register B consists of r2 (low byte) and r3 (high byte).

The index registers are used to store indexes either in memory or in IO space.

Special purpose registers

Flags

The flags register stores state the architecture uses to make decisions. The flags register is 4-bits big.

- Bit 0: ZERO
- Bit 1: EQ (Equals)
- Bit 2: OV (Overflow)
- Bit 3: HALT

Interrupt handler

16-bit register used to store the interrupt handler address.

Stack pointer

16-bit register used to store the stack pointer. The recommended location is at 0xffff.

I/O space mapping

Address	Name
0x0000	GPIO 0 port
0xff00	Interrupt controller
0xff01	Timer 0 prescaler
0xff02	Timer 0 compare
0xff03	Timer 0 control

GPIO

A GPIO port has 8 inputs and 8 outputs. Simply writing to the io address will output the value written to the GPIO port. Reading the io address returns the state of the input pins.

Interrupts

When an interrupt is triggered, the core jumps to the address stored in the interrupt handler register. After the handler has finished executing the routine, the ire instruction is used to return to the previous program code. While an interrupt is being executed, an interrupt lock is set and no other interrupt can occur. The lock is released when the interrupt is finished.

IMPORTANT: The interrupt handler must maintain the state of all registers (general, index and special registers).

Interrupt controller

The interrupt controller contains an interrupt mask that can be set by writing to the IO device. The number of the bit in the interrupt mask corresponds to the interrupt number you want to mask / unmask. Setting a bit to 1 unmasks the interrupt.

During the execution of the interrupt handler, the handler can read the io device to get a mask of the interrupts triggering the interrupt.

Interrupt map

Interrupt number	Function
0	User defined.
1	User defined.
2	User defined.
3	User defined.
4	User defined.
5	User defined.
6	Timer 0 interrupt
7	Software interrupt

Timer

The timer consists of 3 io devices. The prescaler io device is used to set the prescaler, which can change the rate at which the timer counts. The timer will count when prescaler == prescaler_count. The compare io device is used to set the value to which the timer will compare. The timer will trigger the desired action if compare == timer_count. If the prescaler is disabled, the timer will count with the unmodified core clock.

The control io device is used to control the functions of the timer. The following bits can be used:

- Bit 0: if set, timer is enabled.
- Bit 1: if set, prescaler is enabled.
- Bit 2: if set, the timer triggers interrupts.
- Bit 3: if set, the timer pulses the timer output pin.
- Bit 4: when set, the timer toggles the timer output pin.

Instruction set

Arithmetic Operations

- **add <register>, <register | intermediate>**

  Add the left and the right and store in the left register.

  If the addition overflows set the OV flag

- **sub <register>, <register | intermediate>**

  Subtract the right from the left and store in the left register.

  If the subtraction overflows set the OV flag.

- **ado <register>, <register | intermediate>**

  If OV is set add the left and the right and store in the left register.

- **sbo <register>, <register | intermediate>**

  If OV is set Subtract the right from the left and store in the left register.

- **nad <register>, <register | intermediate>**

  Performs logic NAND with the left and the right and stores in left register.

- **nor <register>, <register | intermediate>**

  Performs logic NOR with the left and the right and stores in left register.

- **cmp <register>, <register | intermediate>**

  Compares left and right, resets flags.

  Sets EQ if left and right are equal.

  Sets ZERO if left is equal to zero.

Control flow

- **jeq <index | intermediate>**

  Jump to address if EQ flag is set.

- **jnq <index | intermediate>**

  Jump to address if EQ flag is not set.

- **jzr <index | intermediate>**

  Jump to address if ZERO flag is set.

- **jnz <index | intermediate>**

  Jump to address if ZERO flag is not set.

- **jof <index | intermediate>**

  Jump to address if OV flag is set.

- **jno <index | intermediate>**

  Jump to address if OV flag is not set.

- **jmp <index | intermediate>**

  Jump to address.

Stack operations

- **lsp <index | intermediate>**

  Load stack pointer into stack register.

- **rsp <index>**

  Save stack pointer into index register.

- **put <register | intermediate>**

  Push the value onto the stack.

- **pop <register>**

  Pop from the stack into the register.

Data operations

- **mov <register>, <register>**

  Mov the right register into the left register.

- **lod <register>, <intermediate>**

  Load the intermediate value into the register.

- **out <index>, <register>**

  Save register into io space at index.

- **inp <register>, <index>**

  Load value stored at index in io space into register.

- **wtr <index>, <register>**

  Save register into ram at index.

- **ldr <register>, <index>**

  Load value stored at index in ram into register.

- **lad <index>, <intermediate>**

  Save intermediate into index.

Interrupts

- **ire**

  Return from interrupt.

- **int**

  Trigger software interrupt.

- **lih**

  Load interrupt handler address into interrupt handler register.

Flags

- **cfg**

  Clear all flags.

- **wfg** <register>

  Store register into flags.

- **rfg** <register>

  Load flags into register.

Other

- **nop**

  Do nothing