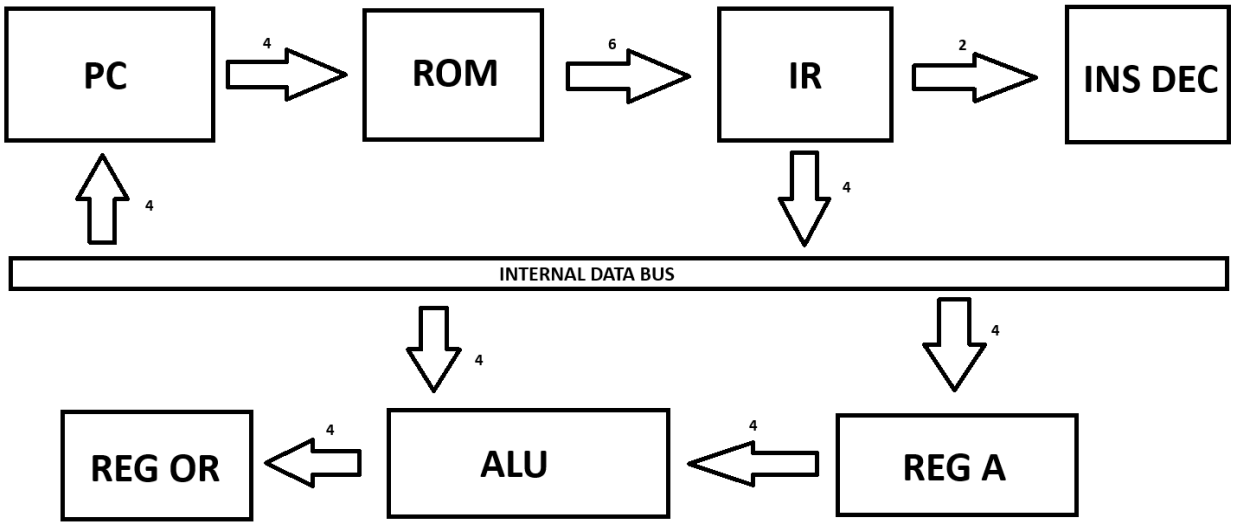
P4 CPU User Manual

Version 1.1

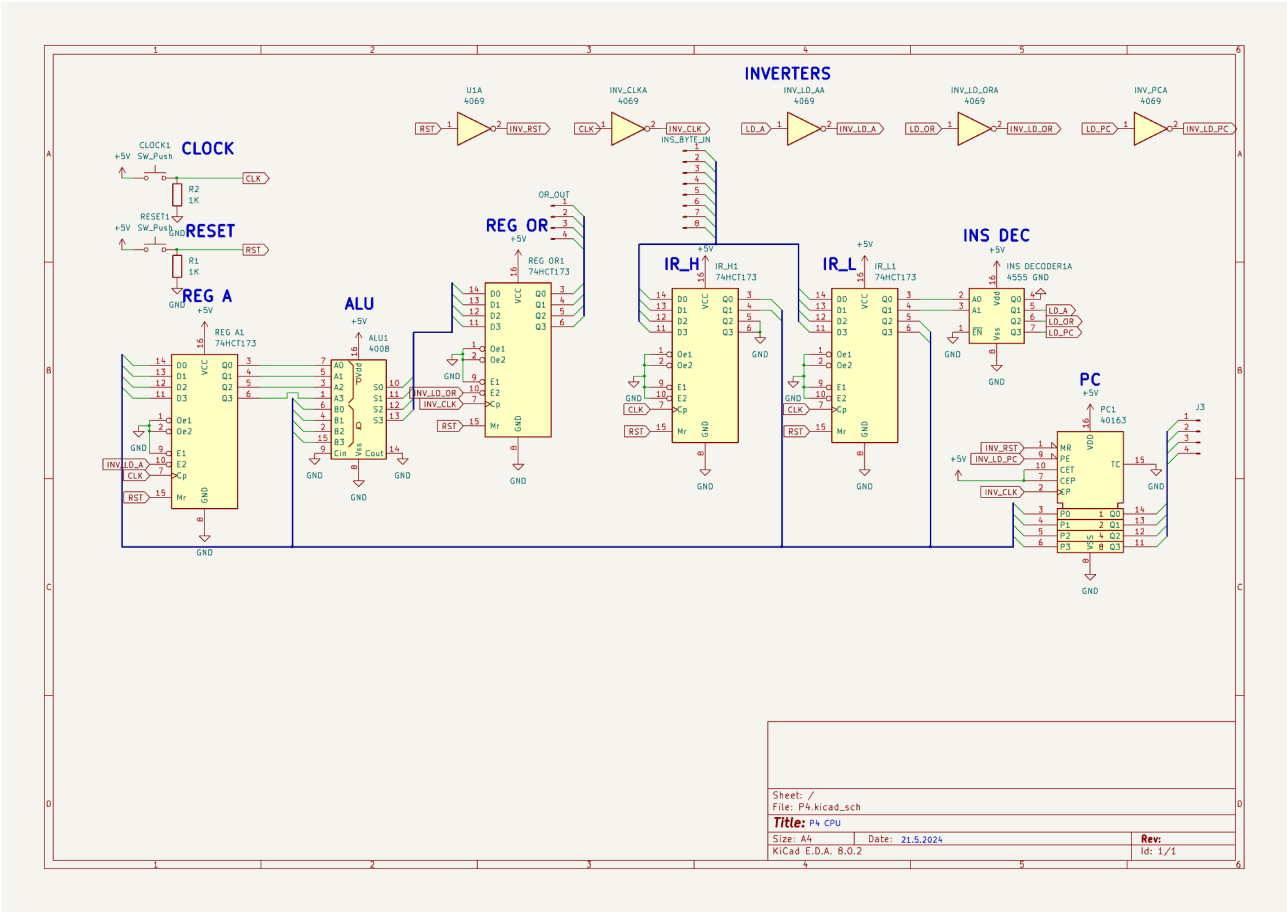
Table of Contents

Architecture



P4 Architecture diagram © Pepe 2024

The P4 or PCPU-1004 is a simple 4-bit processor with a simple single-cycle architecture. It can be made from CMOS or TTL chips. The operating voltage and clock speed vary; +5V is recommended for compatibility with other hardware.



P4 Schematic from integrated circuits.

Registers

The P4 contains two 4-bit general-purpose registers (GPR) and two special-purpose registers (SPR).

General-Purpose Registers

REG A (Accumulator): A 4-bit register that is used to store immediate value for ALU.

REG OR (Output Register): A 4-bit register that is used to store the results of an ALU operation.

Special-Purpose Registers

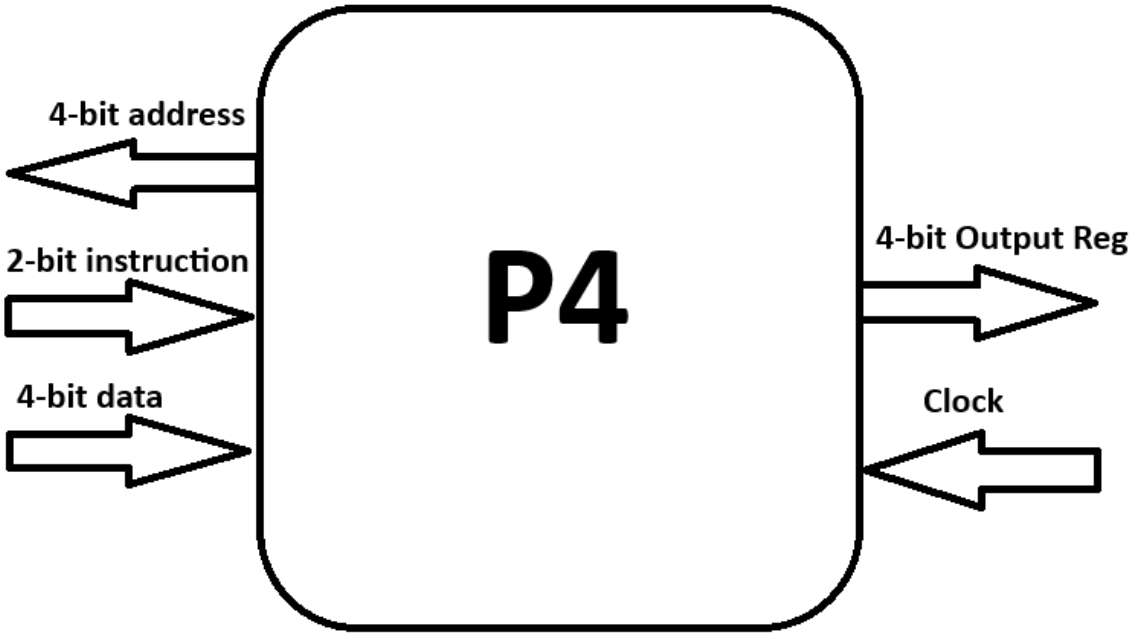
PC (Program Counter): A 4-bit (16 byte program size) or 6-bit (63-byte program size) register that points to the next instruction in the ROM. The PC increments on falling edge. When a jump occurs, the PC is loaded with data in the operand.

IR (Instruction Register): A 6-bit register that holds current instruction fetched from the ROM.

Arithmetic Logic Unit

All the arithmetic and logic operations are performed in the 4-bit ALU. The ALU can do:

- Add



CPU Control

The P4 uses a single-cycle architecture, making the execution fast and simple.

I/O Functions

Address Bus (AB): The address bus carries the value of PC to the memory.

Instruction Bus (IB): The instruction bus carries the instruction from memory.

Data Bus (DB): The data bus carries the data from memory.

Output Register (OR): Can be used for transferring the result to memory or other system components.

Clock (CLK): Clock input to the CPU. The clock synchronizes the registers and other components.

Clock Cycle Operations

Rising edge

- Fetch the instruction byte from ROM.
- Load fetched instruction byte to IR.
- Decode the instruction and create control signals.
- Load data to A if the instruction is LDA

Falling edge

- Execute the instruction.
- Load the result to the OR if the instruction is ADD.
- Increment the PC.

P4 Instructions

The P4 uses Reduced Instruction Set Computing (RISC), reducing the complexity of the architecture and instructions.

Instruction Byte

The Instruction Byte (IB) contains a 4-bit operand and 2-bit instruction. The IB can fit into a single 8-bit memory address. If the PC is 6-bit, then the last two empty bits in the IB are used for address.

INSTRUCTION BYTE

EMPTY	EMPTY	Operand3	Operand2	Operand1	Operand0	Opcode1	Opcode0
Bit 7							Bit 0

Instruction Set

The P4 has 4 different instructions. Every instruction takes a single clock cycle to complete.

P4's Instruction Set:

NOP

Operation: No Operation

Operands: None

Opcode: 0b00, 0x00

LDA n

Operation: Load A with Immediate value

Operands: n

Opcode: 0b01, 0x01

ADD n

Operation: Adds A with Immediate value to OR

Operands: n

Opcode: 0b10, 0x02

JMP n

Operations: Unconditional jump, loads PC with Immediate value

Operands: n

Opcode: 0b11, 0x03

Programming the P4 CPU

The P4 supports up to 63 bytes (at 6-bit PC) of memory, so program space is more than the default 16 bytes (at 4-bit PC). Programs can be written in assembly or machine code. When written in assembly, the code needs to be assembled into machine code. Machine code needs to be in hex format when writing to an EEPROM.

Programming in assembly

The P4 uses simple assembly language. It's way easier to write programs in assembly than in machine code. The P4 assembly needs a custom assembler.

PASM Assembler

P4 uses Pepe's Assembler (PASM) for assembling the program to machine code. The assembler doesn't support labels!

Installation steps

1. Create a file and named it to *pasm.cpp* and copy the code from the below.

2. Save the file and compile it, example: *g++ pasm.cpp -o pasm.cpp* .

3. Create an assembly file to same folder as the *pasm.cpp* and write your program there.

4. Assemble the program, example: *./pasm -c test.asm -o test.bin -HEX* . If you want binary format, change *-HEX* to *-BIN* .

```
#include <iostream> // include all the necessary libraries
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
#include <bitset>
#define helpMessage "-c <source file> -o <output file> -h help -v version -BIN binary output -HEX hex output"
#define version "Pepe's assembler v1.0"
using namespace std;
vector<string> program; // source program
vector<string> assembledProgramBIN; //binary program
vector<string> assembledProgramHEX; //hex program
bool format = false; // 0 - binary, 1 – hex
string filename;
ifstream source_file;
ofstream output_file;
string dectohex(int num) {
   stringstream ss;
   ss << hex << num;
   return ss.str();
}
```

```
void p4() {
    string line;
    while(getline(source_file, line)) {
        program.push_back(line);
    }
    for(int i = 0; i < program.size(); i++) {
        if(program[i].find("nop") == 0 | program[i].find("NOP") == 0) {
            assembledProgramBIN.push_back("00000000");
        }
        if(program[i].find("lda ") == 0 | program[i].find("LDA") == 0) {
            int imm = stoi(program[i].substr(4));
            string binaryImm = bitset<6>(imm).to_string();
            assembledProgramBIN.push_back(binaryImm+"01");
        }
        if(program[i].find("add ") == 0 | program[i].find("ADD") == 0) {
            int imm = stoi(program[i].substr(4));
            string binaryImm = bitset<6>(imm).to_string();
            assembledProgramBIN.push_back(binaryImm+"10");
        }
        if(program[i].find("jmp ") == 0 | program[i].find("JMP") == 0) {
            int imm = stoi(program[i].substr(4));
            string binaryImm = bitset<6>(imm).to_string();
            assembledProgramBIN.push_back(binaryImm+"11");
        }
    }
    for(int i = 0; i < assembledProgramBIN.size(); i++) {
        if(format == 0) {
            output_file << assembledProgramBIN[i] << endl;
            cout << assembledProgramBIN[i] << endl;
        } else if(format == 1) {
            output_file << dectohex(stoi(assembledProgramBIN[i], nullptr, 2)) << endl;
            cout << dectohex(stoi(assembledProgramBIN[i], nullptr, 2)) << endl;
        }
    }
}
int main(int argc, char *argv[]) {
    for(int i = 0; i < argc; i++) {
        if (argc == 1) {
            cout << "Error: No arguments" << endl;
            cout << “Usage: ” << helpMessage << endl;
            return 1;
        }
        if(string(argv[i]) == "-h") {
            cout << helpMessage << endl;
```

```
    }
    if((string(argv[i])).find("-c") == 0) {
      filename = argv[i+1];
      source_file.open(filename);
    }
    if((string(argv[i])).find("-o") == 0) {
      filename = argv[i+1];
      output_file.open(filename);
    }
    if(string(argv[i]) == "-v") {
      cout << version << endl;
    }
    if(string(argv[i]) == "-BIN") {
      format = 0;
    }
    if(string(argv[i]) == "-HEX") {
      format = 1;
    }
  }
  if (source_file.is_open() && output_file.is_open()) {
    p4();
    source_file.close();
    output_file.close();
  }
  return 0;
}
```

Assembly examples

Example 1, Simple addition

Here's a program that loads 1 to A and adds 1 and then jumps to address 0:

```
LDA 1
ADD 1
JMP 0
```

In machine code (Binary):

```
00000101
00000110
00000011
```

In machine code (Hex):

```
5
6
3
```

Example 2, Bit shifter loop

Here’s a program that shifts a single bit in a loop:

LDA 1

ADD 0

ADD 1

ADD 3

ADD 7

ADD 15

ADD 7

ADD 3

ADD 1

ADD 0

JMP 1

In machine code (Binary):

00000101

00000010

00000110

00001110

00011110

00111110

00011110

00001110

00000110

00000010

00000111

In machine code (Hex):

5

2

6

e

1e

3e

1e

e

6

2

7