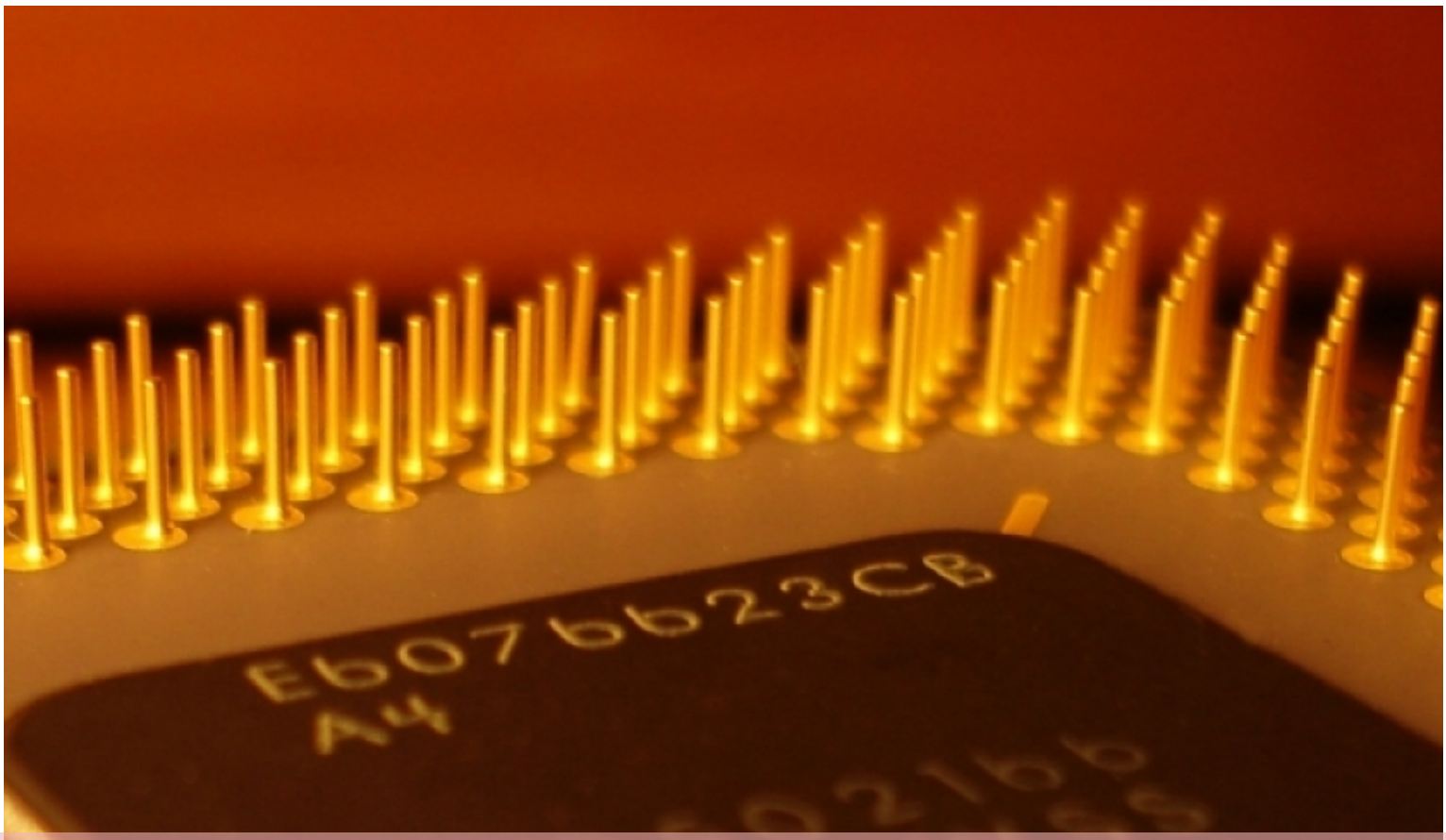


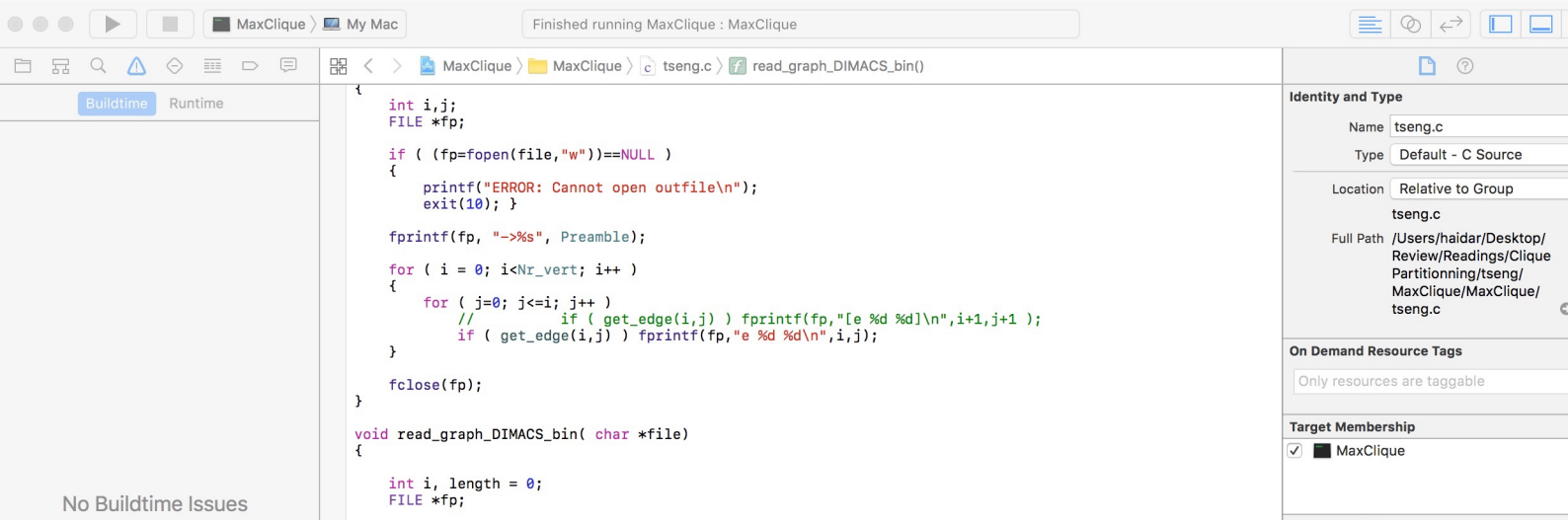
Hands-On Computer Organization Lab

With C and Verilog

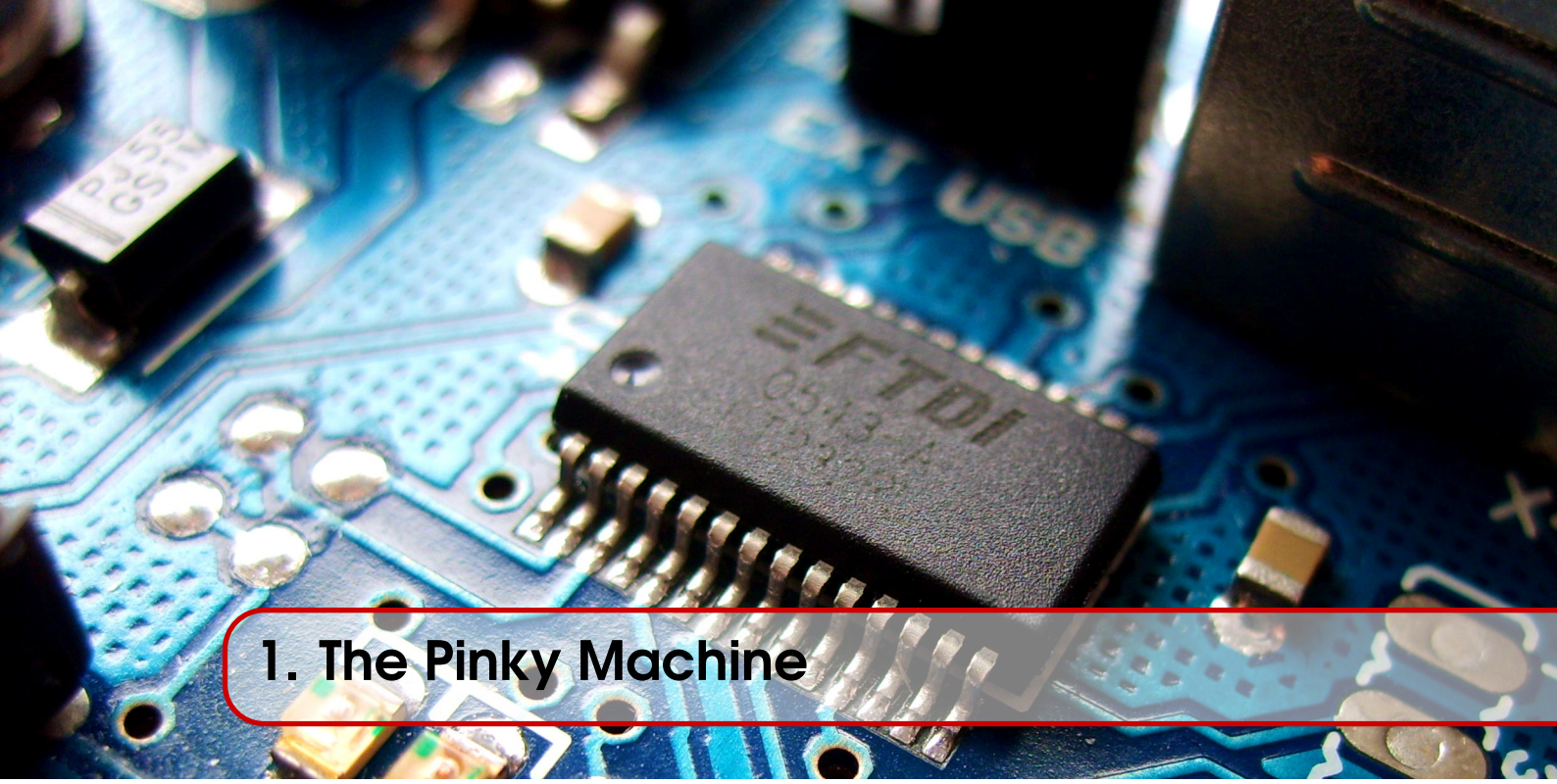
Haidar M. Harmanani

HARMANANI.GITHUB.IO

First printing, March 2016



Contents



1. The Pinky Machine

"RISC Architecture is Gonna Change Everything"
—Hackers (1995)

1.1 The Von Neumann Architecture

At higher-levels of abstraction, a single processor is described as a *von Neumann architecture* with a single memory that stores a program and its data. The processor executes the program using a *fetch*, *execute*, and *store* cycles. A *von Neumann architecture* would typically include instructions to load data from the memory to the registers, to store data that are in registers in the memory, and an array of logical and arithmetic instructions. A simple execution model would be as follows:

1. *Instruction fetch (IF)*: the next instruction that is indexed by the program counter (or instruction pointer) is loaded into the processor.
2. *Instruction decode (ID)*: The decoding unit in the processor inspects the instruction to determine the operation and the operands.
3. *Instruction Execution (EX)*: The operation is executed, reading data from registers and writing it back to a register.
4. *Memory (MEM)*: The memory access stage where necessary data is brought from memory into a register.
5. *Write Back (WB)*: The register content is written back to memory.

1.2 The Pinky Machine

In this lab, we would look at the design and simulation of a very simple ARM processor that we will coin "the pinky processor." The `Pinky` processor is a 16-bit RISC-based architecture that is based on the ARM processor. The `Pinky` processor has a 256K memory and 8 16-bit registers. In order to simplify the implementation, we will assume that the first generation `Pinky` processor has two memories: a *data memory* and an *instruction memory*. The data memory is used in order to

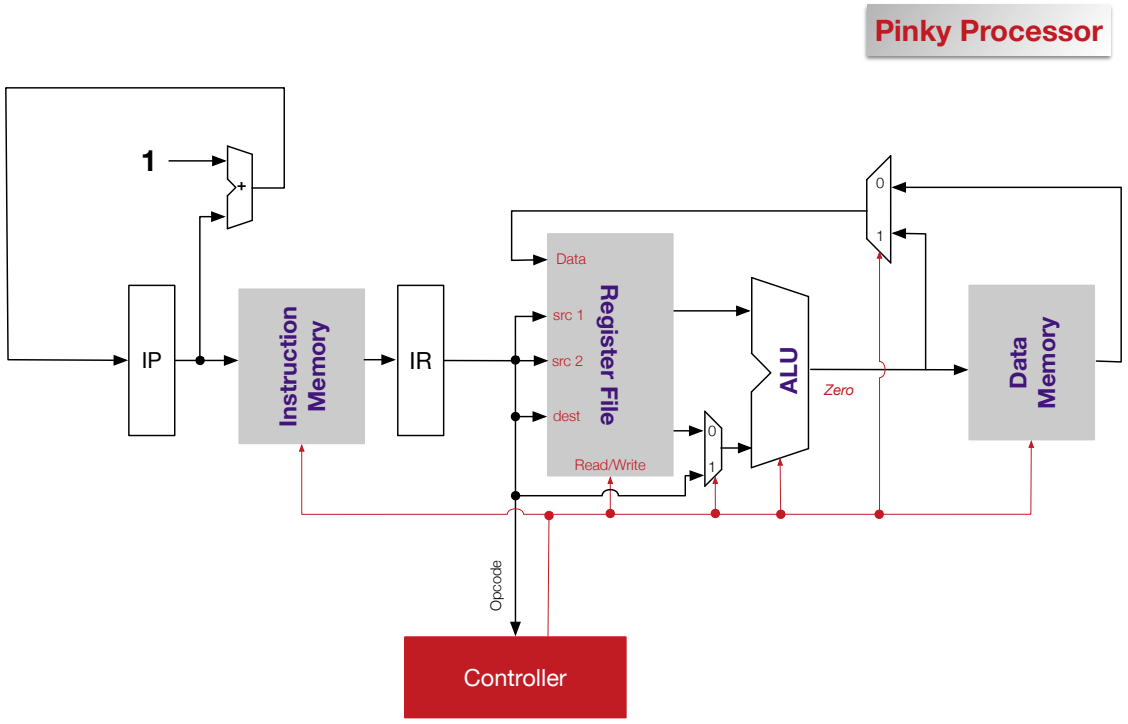


Figure 1.1: Data path for the Pinky Machine

store the user's data while the instruction memory is used the to store the user's program. The 8 general purpose registers are labeled `W0-W7` and are arranged into a register file.

Table 1.1 illustrates the `Pinky` processor instruction set architecture which includes the following family set ISA:

1. The `D-Type` which refers mostly to `load` and `store` instructions.
2. The `R-Type` which refers to register to register instructions. These include: `ADD`, `SUB`, `AND`. `OR`, `XOR`, `ADDS`, `SUBS`, `SL`, and `SR`.
3. The `I-Type` which refers to immediate instructions including: `ADDI` and `SUBI`.
4. The `CB-Type` which refers to conditional branch instructions including: `CBZ` and `CBNZ`
5. The `B-Type` which refers to branch instructions such as `BR`. The *Branch and Link* instruction, `BL` is used in order to call a subroutine. The return address is saved in register `W30` and is used to restore the value of the PC when a `RET` instruction is executed.
6. The `Halt-Type` which refers to one instruction, `Halt`, which is used to stop the program execution.

The D-Type instructions will load data from the data memory and store data in the data memory using two instructions, *LDW* and *STW*. The *R-Type* instructions execute instructions. The first generation `Pinky` processor includes *add*, *sub*, *or*, *and*, *xor*, and *shift* instructions. The *opcode* of the `R-Type` instruction determines whether *source 1* (Figure 1.2) is a register or an immediate shift amount.

Programs are stored in the memory and are indexed by the program counter (PC). All instructions are 16 bits in length, and are word aligned. The `Pinky` instructions are made to execute conditionally by postfixing them with the appropriate condition code field:

```
ADD(W1, W2, W3)      // address 11
SUB(W0, W1, W2)     // address 12
```

OpCode	Instruction	Mnemonic	Effect
0	HALT	HALT	Halts execution
1	LDW	LDW(W_a, address)	PC++; $W_a \leftarrow$ M[address]
2	STW	STW(W_a, address)	PC++; M[address] $\leftarrow W_a$
3	ADD	ADD(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] + R[W_c]
4	SUB	SUB(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] − R[W_c]
5	AND	AND(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] & R[W_c]
6	OR	OR(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] | R[W_c]
7	XOR	XOR(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] $\oplus$ R[W_c]
8	ADDS	ADDS(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] + R[W_c]; $Z \leftarrow (R[W_a] == 0)\ ?\ 1\ :\ 0$ $N \leftarrow (R[W_a] < 0)\ ?\ 1\ :\ 0$
9	SUBS	SUBS(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] − R[W_c] $Z \leftarrow (R[W_a] == 0)\ ?\ 1\ :\ 0$ $N \leftarrow (R[W_a] < 0)\ ?\ 1\ :\ 0$
10	ANDS	ANDS(W_a, W_b, W_c)	PC++; R[W_a] = R[W_b] & R[W_c] $Z \leftarrow (R[W_a] == 0)\ ?\ 1\ :\ 0$ $N \leftarrow (R[W_a] < 0)\ ?\ 1\ :\ 0$
11	SL	SL(W_a, W_b, $ShiftAmt$)	PC++; R[W_a] = R[W_b] << ShiftAmt
12	SR	SR(W_a, W_b, $ShiftAmt$)	PC++; R[W_a] = R[W_b] >> ShiftAmt
13	ADDI	ADDI(W_a, W_b, Imm)	PC++; R[W_a] = R[W_b] + Imm
14	SUBI	SUBI(W_a, Imm)	PC++; R[W_a] = R[W_b] − Imm
15	BR	BR(`Address`)	PC $\leftarrow$ `Address`
16	BZ	BZ(`Address`)	if (Z == 1 then PC $\leftarrow$ Address; else PC++
17	CBZ	CBZ(W_a, `Address`)	if (W_a == 0) then PC$\leftarrow$ `Address`; else PC++
18	CBNZ	CBNZ(W_a, `Address`)	if (W_a != 0) then PC $\leftarrow$ `Address`; else PC++
19–128	Not Used		

Table 1.1: Pinky Processor Instruction Set Summary

1.2.1 Condition Codes

The pinky machine has two flags N and Z that are set by the outcome of the `ADDS`, `SUBS`, `ADDIS`, and `SUBIS` instructions as follows:

- **N == 0**: The outcome is greater or equal to 0, which is considered positive, and so the N (negative) bit is set to 0.
- **Z == 1**: The outcome is 0, so the Z (zero) bit is set to 1.

Thus, `ADDS W0, W1, W2` is equivalent to `W0 = W1 + W2` and sets the Z flag to 1 if W0 is 0 and the N flag to 1 if W0 $\geq$ 0. The flag register is also used to transfer control using the `BZ` instruction.

1.2.2 Control Transfer

The pinky machine provides four instructions for control transfer: `BR`, `BZ`, `CBZ`, and `CBNZ`. The `BR` performs an unconditional control transfer by branching to the target address while `BZ` performs a control transfer if the Z flag is 1. In either case this is accomplished by adding the `offset` to the PC. Finally, the `CBZ` compare and branch on zero while and `CBNZ`, on the other hand compare and branch on non-zero.

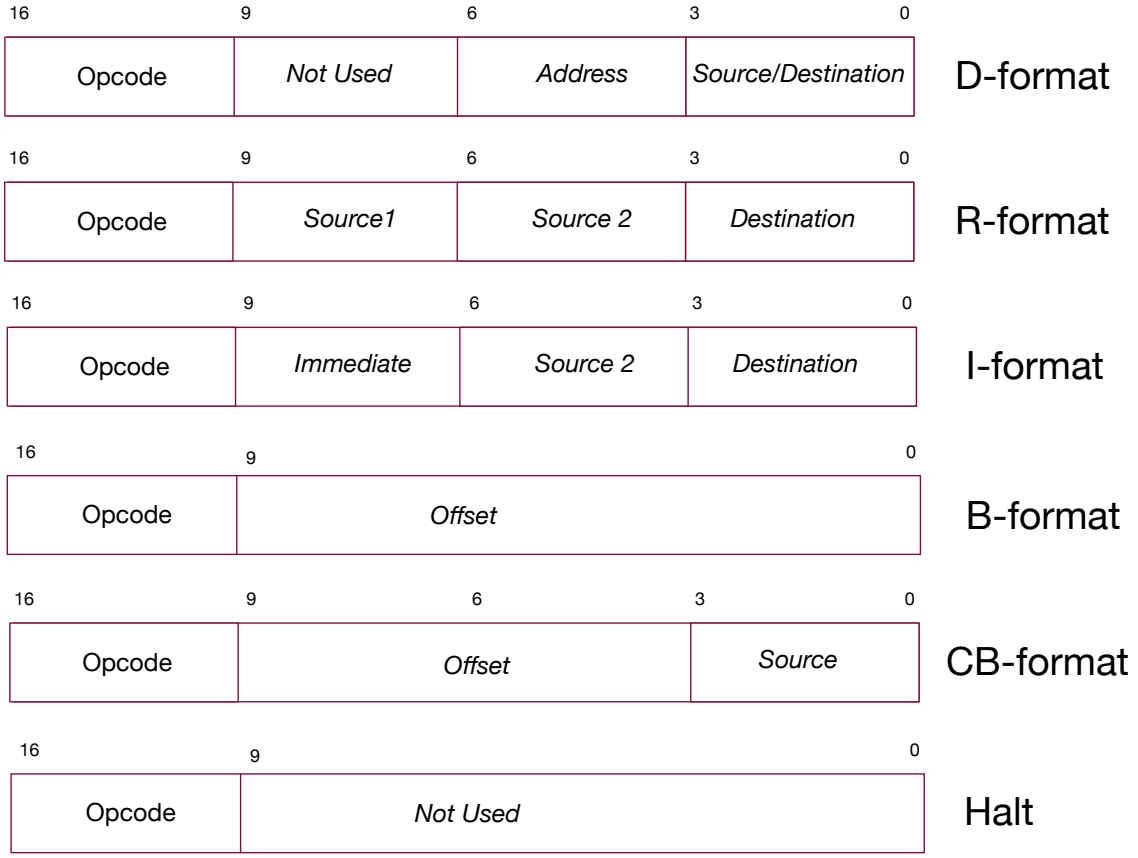


Figure 1.2: Pinky Processor Instruction Set

1.3 C Elements for an Instruction Set

There are four elements that you would need to implement in order to simulate an instruction set. These are as follows:

1. Opcodes
2. Mnemonics
3. Hardware components including memory, registers, ALUs, and interconnection components

This lab focuses on the implementation of the `Pinky` processor shown in Table 1.1 and in Figure 1.2.

1.4 Lab

The lab this week involves developing a complete C-model for the `Pinky` processor. You are to work alone and you will not get any help. You will be supplied with a working C skeleton. Please read the code and ask any questions that you may have prior top the lab. Then complete all missing instructions in the program. That includes the instructions as well as the code. Please also modify the fetch and decode unit. The following C constructs may help while developing your code:

1. Define the Opcodes

```
#define OP_HALT 0x00
#define OP_LD   0x01
#define OP_ST   0x02
#define OP_ADD  0x03
#define OP_SUB  0x04
#define OP_AND  0x05
```

```
...
#define W31 0x1F
```

2. Create the Mnemonics

```
#define LOAD(DestReg, Address)           (OP_LD  << 9) | (Address << 3)  | DestReg
#define STORE(SourceReg, Address)        (OP_ST  << 9) | (Address << 3)  | SourceReg
#define ADD(Reg1,Reg2,Reg3)              (OP_ADD << 9) | (Reg1 << 6) | (Reg2 << 3) | Reg3
#define AND(Reg1,Reg2,Reg3)              (OP_SUB << 9) | (Reg1 << 6) | (Reg2 << 3) | Reg3
#define HALT                             OP_HALT) | Source
#define ADD(Dest, Source2, SourceReg1)  (OP_ADD << 20) | (Source2 << 15) | (Source1 << 5) | Dest
```

3. All instructions are stored in memory using 16 bits. Thus, instructions have to be decoded in order to decipher the operations as follows:

```
unsigned short InstructionDecodeUnit(unsigned short uInstruction, unsigned short *uDest,
                                     unsigned short *uReg1, unsigned short *uReg2)
{
    unsigned short uOpCode = (uInstruction >> 9) & 0x3F;

    switch (uOpCode) {
        case OP_LD:
        case OP_ST:
            *uDest = (uInstruction >> 9) & 0x0F;
            *uReg1 = (uInstruction & 0x0F);
            *uReg2 = 1;                              // not used
            break;

        case OP_ADD:
        case OP_SUB:
            *uDest = (uInstruction >> 9) & 0x0F;
            *uReg2 = (uInstruction >> 6 & 0x0F);
            *uReg1 = (uInstruction & 0x0F);
            break;

        case OP_HALT:
            break;

    }

    return(uOpCode);

}
```

4. Use the following values in the data memory:

```
unsigned short DM[64] =
{
    5,              // Address: 0
    10,             // Address: 1
    7,              // Address: 2
    20,             // Address: 3
    0               // Address: 4
};
```

5. Test your code by executing the following:

```
LOAD(W0,3),     // 0
LOAD(W1,2),     // 1
SL(W2, W1,1),        // 2
ADD(W3,W0,W2),  // 3
STORE(W4, 5),   // 4
LOAD(W5,3),     // 5
SUB(W6,W5, W2), // 7
```

```
AND(W7, W6,W2),  // 8
SR(W0,W7,1) // 9
```

Lab 1.1 Complete the design of the `Pinky` processor by:

1. Implementing all instructions that are shown in Table 1.1;
2. Test all instructions and ensure that they are working properly;
3. Ensure that the above program is properly implemented. What is the answer in hex?
4. Upload your work on `Google Classroom` along with the report.

■