Agriculture System Chip

Internet of things instruction set architecture

Table of Contents

1. Group member details…… [2]
2. Register Profile…… [2]
3. Operands and operations…… [3]
4. Instruction Encoding…..…….. [4]
5. Instruction Format…….. [5]
6. Performance Requirements…….. [5]
7. Instruction set architecture rationalization…… [7]
8. Appendix……[9]

Group members details

- Liteboho Maseli – 202101659
- Hlalele Manganye – 202101681
- Khoase Hashatsi – 202101671
- Hlalefang Makhasane – 202002398
- Keketso Tolo – 202100092

Instruction Set Architecture for Internet of Things Agricultural System

Register Profile

General purpose registers:

- Temporary registers: 4 ($t0 - $t3)
- Saved registers: 4 ($s0 - $s3)
- Return value registers: 2 ($v0, $v1)
- Zero register: 1 ($zero)
- First four arguments to functions register: 4 ($a0 - $a3)
- Assembler register: 1($at)

Given the relatively simple nature of the internet of things task specification, these 16 general purpose registers should be more than sufficient. They reduced number also aids in the overall reduction of the size and price of the chip.

Special Purpose Registers:

- Program Counter (pc)
- Stack Point Register (sp)
- Status Register (sr)
- Frame Pointer (fp)

The above are meant for the managing of the internet of things specific functions e.g., the move to next instruction found in a certain address; this address is stored in the program counter.

Operations

Arithmetic and logic:

- ADD
- DIV
- AND
- OR
- NOT

Given the simple nature of the computations that our ISA needs to perform i.e., the summation of the values captured from the sensors then being averaged, the main arithmetic operations will then need to be addition to calculate the sum, division for calculating the average. As for comparison (if value x < y, execute instruction z), we will need different operands such as the branch and jump operands. For data processing, all the basic gates will be used for this.

Other operations:

- Branch
- Jump
- Load
- Store

The branch is meant for the execution of conditional statements. For the piece of code where we compare the value of moisture to the set value that is required, the branch operand will aid in the execution of going to the next instruction that needs to be executed i.e., if y <70, branch to instruction that says add more water. These operations are all key to the control flow of this system. The jump operations will be used whenever we must loop through an array to compute the sum of all readings captured from the sensor.

Addressing Modes

1. **Immediate Addressing**: In this mode, the operand value is directly specified within the instruction itself. For example, ***"addi $t0, $t1, 10"*** adds immediate value 10 to the contents of register $t1 and stores the result in $t0.
2. **Register Addressing**: In this mode, the operand is specified using a register. For example, "add $t0, $t1, $t2" adds the values of registers $t1 and $t2, and stores the result in $t0.
3. **Base Addressing**: This addressing mode is used in memory instructions. It uses a base register value and an offset to access data in memory. For example, ***"lw $t0, 100($t1)"*** loads a word from memory at the address specified by the sum of $t1 and 100, and stores it in $t0.
4. **PC-relative Addressing**: In this mode, the operand is specified relative to the current program counter (PC) value. It is mainly used for branching instructions. For example, ***"beq $t0, $t1, label"*** branches to the instruction labeled "label" if the values of $t0 and $t1 registers are equal.

Instruction Encoding

Op Code (6 bits)	Instruction
000000	ADD
000001	DIV
000010	AND
000100	BRANCH
001000	JUMP
010000	LOAD

Explanations: We will be using binary encoding, and we will be using big endian so the first 6 bits of code is the operant to be performed and 5 bits to represent the registers to store, load and do calculations.

Instruction Format

I-type (immediate type) which is the data, branch instruction format.

Op code	rs	rt	Immed
6 bits	5 bits	5 bits	8 bits

R-type (register type) which is the arithmetic instruction

Op Code	rs	rt	rd	Shit
6 bits	5 bits	5 bits	5 bits	3 bits

J-type (register type) which is the arithmetic instruction

Op Code	Adress
6 bits	18

Where;

- rs – register source
- rt – register source
- rd – register destination

Performance Requirements

The performance requirements of IoT chips are crucial in meeting the real-time demands of IoT applications:

- **Processing Power**: IoT applications often require real-time data processing and analysis. Therefore, the ISA should have sufficient processing power to perform complex calculations and handle high data throughput.

- **Low Latency:** Many IoT applications require real-time response and low latency. Our ISA provides efficient and fast execution of instructions to minimize delays and ensure quick response times.

- **Energy Efficiency**: IoT devices are powered by batteries or have limited power sources. Our ISA is designed to minimize power consumption, enabling devices to operate for extended periods without frequent battery replacements.

- **Memory Management:** IoT applications generate and handle large amounts of data. The ISA should have efficient memory management capabilities to handle data storage, retrieval, and manipulation with minimal overhead.
- **Security**: The ISA includes security features such as encryption, authentication.

- **Connectivity**: IoT devices need to communicate with each other and with cloud platforms. The ISA should support various connectivity options like Wi-Fi, Bluetooth, etc., along with the necessary protocols.

- **Real-Time Task Scheduling**: IoT applications rely on timely execution of tasks or events. The ISA facilitates efficient real-time task scheduling and ensure accurate timing and synchronization among devices.

- **Reliability and Fault Tolerance**: IoT devices operate in diverse environments and undergo various environmental stresses. The ISA provides resilience against faults, ensure error detection and recovery mechanisms, and offer high reliability in data processing.

Instruction set architecture rationalization

- **Register Profile**: A moderate number of registers balances flexibility and hardware complexity.
  We have decided to include:
  1. **$zero register**: which is always a zero constant. It aids us when returning function calls e.g., (return 0;) in c++
  2. **$at** : register reserved for assembler
  3. **$v0-$v1**: these are our return value registers, that return results obtained after function calls.
  4. **$t0-$t3**: these are registers used to temporarily store data when carrying out computations.
  5. **Store/save registers**: e.g., ($s0), these registers are used to store values longer than temporary registers, which are handy when variables are required in different functions.

- **Operations**: The selected operations cater to data processing and IoT-specific needs while keeping the ISA relatively simple.
  1. *Arithmetic instructions*:
     a) The add instruction such as (add $s0, $s0, $s2) will be used when computing the sum of the values of our sensors installed on a certain plot, to enable their average evaluation.
     b) *The divide instruction*: this is an instruction of the form (div $t0, $t0, $s0) which will be useful when computing the average of our data from sensors.
  2. *Data transfer instructions*:
     These are instructions used to transfer data from the memory of our IoT microprocessor to the registers. Values stored in memory can be values such as the comparison value. For example, the statement (if (a <70), do something) where a in this case is the average from sensors and 70 is the comparison value.
  3. *Conditional branch instructions*:
     They are used when executing comparisons.
     Operations:
     a) *beq*: used when comparing IoT value in memory and data from sensors that are stored in registers. True if both values are equal.
     b) *bne*: inverse operation to beq.

c) *slt*: used to check the greater value between data stored in registers. This will determine when our IoT device sends an output signal to water the plots and when to stop watering.

4. *Unconditional jump*:
   Operation: a) j: jumps to target address

- **Addressing Modes**: Our Instruction Set Architecture (ISA) for an IoT (Internet of Things) device uses multiple addressing modes to enhance the device's versatility and efficiency in handling a variety of tasks.
  These modes allow for versatile memory access while maintaining simplicity.
  a) *Immediate Addressing*: immediate addressing is used to specify data directly within instructions. It's efficient for small, constant values.
  b) *Direct Addressing*: Direct addressing involves specifying the memory address of an operand directly in the instruction. It's useful for accessing variables or data stored in memory.
  c) *Indirect Addressing*: In this mode, the instruction points to an address in memory, which, in turn, holds the actual data. It provides flexibility when dealing with data structures or when the specific memory address needs to be calculated.
  d) *Register Addressing*: Our ISA allows direct access to data stored in registers. This is often the fastest and most power-efficient addressing mode.
  e) *Stack Addressing*: IoT devices may use stack-based addressing for managing function calls and local variables.

Appendix

Sample C++ code for the ISA implantation

```
#include <iostream>
struct SensorData {
    int* readings; // pointer to an array of readings int numReadings;
};
// Function to compute the average reading from sensors
double computeAverage(SensorData data) {
    int sum = 0;
    for (int i = 0; i < data.numReadings; i++) {
    sum += data.readings[i];}
    return (double)sum / data.numReadings;
}
int main() {
    int readingsArray[5] = {72, 68, 70, 73, 69};
    SensorData data = {readingsArray, 5};
    double averageReading = computeAverage(data);
    if (averageReading < 70.0) {
        std::cout << "Watering needed!" << std::endl;
    }
else {
    std::cout << "No watering needed." << std::endl;
}
    return 0;
}
```