ISA Design for the IoT

This will be a RISC ISA because the IoT computes simple arithmetic operations and the RISC approach allows for a more straightforward design. Here are its specifications:

Register Profile

1. Reduced Register File:

We have opted to reduce the number of registers in the register file for several reasons:

a) Cost reduction

b) Sequential workload

c) Improved processor speed due to reduced complexity

d) Shorter instruction encoding for operands in registers

e) Decreased processor complexity

2. Saved Registers:

After analyzing our C++ code, we have determined that only a subset of registers is needed. Consequently, we have reduced the number of saved registers from 8 to 3, with only s0, s1, and s2 retained.

3. General Purpose and Special Registers:

The remaining general-purpose and special registers will continue to be part of the register file, maintaining the same addressing methods as in a typical MIPS32 ISA.

4. Register Size:

The size of each register remains 32 bits, ensuring compatibility with the same operand lengths as a basic MIPS32 ISA.

Operations

1. Arithmetic Operations:

Our RISC ISA supports only fundamental arithmetic operations, including addition, subtraction, division(/) through addition and subtraction, and less-than(<) comparisons—all of which are implemented through addition and subtraction. This design choice simplifies the ISA and enhances processing efficiency by using straightforward operations at the core of the processor.

Implementation of division(/)

This only explains how we envision the division process will work and is not the final draft of the division for our ISA:

sum=0

count=0

for all numbers in the array:

sum= sum + num

count= count+1

average = sum

for i from 1 to count

average =average – count

And so for as long as average is not negative or is not 0, average=average-count.

Implementation of less-than(<)

x-70.0

if (x-70.0) is negative then x is less than 70.0 , if it is 0 then x=70.0 and if it is positive then x is greater than 70.0

Sample Arithmetic Operation Instructions:

1. Addition (add) Instruction:

- Format: add Rd, Rs1, Rs2
- Example: add $t0, $s1, $s2
- Implementation: Rd = Rs1 + Rs2

2. Subtraction (sub) Instruction:

- Format: sub Rd, Rs1, Rs2
- Example: sub $t1, $s2, $s0
- Implementation: Rd = Rs1 - Rs2

3. Division (div) Instruction:

- Format: div Rd, Rs1, Rs2
- Example: div $t0, $s1, $s2
- Implementation:

```
# sum = 0
# count = 0
# for all numbers in the array:
# sum = sum + num
# count = count + 1
# average = sum
# for i from 1 to count:
# average = average - count

# Assuming $s1 has the sum and $s2 has the count
div $t0, $s1, $s2                    # average = sum / count
```

4. Less-than(lt) Instruction:

- Format: lt Rd, Rs1, Rs2
- Example: lt $t1, $s0, $t2
- Implementation:

```
# x - 70.0
sub $t3, $s0, $t4

# if (x - 70.0) is negative then x is less than 70.0,
```

if it is 0 then x = 70.0,

and if it is positive then x is greater than 70.0

lt $t1, $t3, $zero # $t1 = 1 if (x - 70.0) is negative, 0 otherwise

2. Control Flow Operations:

The ISA accommodates two control flow operations: jump instructions for function calls and branch instructions for if statements.

Sample Control Flow Instructions:

1. Jump (j) Instruction:

- Format: j target
- Example: j loop_start

2. Branch (b) Instruction:

- Format: b Rs, Rt, target
- Example: b $s1, $s2, condition_true
- Implementation: Branch to `target` if the condition specified by `Rs` and `Rt` is true.

3. Memory Operations:

1. Load (lw) Instruction:

- Format: lw Rd, offset(Rs)
- Description: Loads a word from memory into a register.
- Example: lw $t0, 4($s0)
- Implementation: Rd = Memory[Rs + offset]

2. Store (sw) Instruction:

- Format: sw Rs, offset(Rd)

- Description: Stores the contents of a register into memory.

- Example: sw $s1, 8($s2)

- Implementation: Memory[Rd + offset] = Rs

Addressing Modes

We have chosen to employ two primary addressing modes:

- **Register Direct:** This mode is ideal when operands (values in the array) are stored in registers. It allows these values to be directly accessible for function execution, reducing memory access overhead and promoting faster processing.

  Example: lw $t0, 4($s0)

  Description: Directly accesses values stored in registers, minimizing memory access overhead.

- **Register Indirect:** In cases where the limited number of registers necessitates frequent memory access, this mode facilitates the storage of operand addresses in registers, while the operands themselves remain in main memory. This approach incurs some added expense and time but ensures greater flexibility.

  Example: lw $t0, 4($t1)

  Description: Stores operand addresses in registers while operands reside in main memory.

Ultimately, our ISA design leverages the Register Direct addressing mode to achieve the desired performance characteristics.

Instruction Format and Encoding

The instruction format of our ISA mirrors the MIPS32 ISA format, characterized by a 6-bit opcode field, source and destination register fields, an immediate value field, and a function code field. The encoding process uses these fields to determine the operation and operand information. Also, our ISA will not include any immediate values as there will not be a need for them. The only immediate value that could

be used in our IoT is 70.0 used for comparisons but we have decided to store this 70.0 in a register to avoid the process of having to change the 70.0 into 32bit format when it is called and so this helps in making processing faster.

Instruction Format

- Opcode (6 bits): Specifies the operation to be performed.

- Source Register (5 bits): Specifies the source register.

- Destination Register (5 bits): Specifies the destination register.

- Immediate Value/ Register (5 bits): Can be used for an immediate value or to specify register holding a constant (e.g., the register holding 70.0) but the filed will be otherwise unused if the register holding 70.0 is not being used.

- Function Code (5 bits): Further refines the operation, especially for arithmetic operations.