Brayden-Vidhur Processing Unit (BVPU)

Authors:Brayden Abo and Vidhur Busannagari

Contributions:

- Brayden Abo: Primary focus on hardware design, including developing the CPU circuits in Logisim-Evolution.
- Vidhur Busannagari: Main contributor to the assembler code and CPU instruction encoding.
- Collaboration: Both authors contributed equally to this report and overall testing of the CPU for each other's parts as well as our own.

1. Overview

The **Brayden-Vidhur Processing Unit (BVPU)** is a custom-designed CPU created using **Logisim-Evolution**. It features a simplified but functional instruction set architecture (ISA) tailored for fundamental arithmetic operations and memory management. BVPU supports a small set of instructions, implemented in both register-to-register and register-to-immediate formats.

---

2. Using the Assembler

The assembler is implemented in Python and converts BVPU assembly language instructions into machine code. Here's how to use it:

2.1 Dependencies

Ensure Python is installed on your system. The assembler uses standard Python libraries and requires no external dependencies.

2.2 Assembler Usage

To assemble a program:

```
python assembler-1.py <input_file> <output_file>
```

- `<input_file>`: The text file containing your BVPU assembly code.
- `<output_file>`: The file where the assembled machine code (in hexadecimal) will be saved.

2.3 Loading into Memory

1. Use the output of the assembler to initialize instruction memory in Logisim-Evolution.
2. The assembler generates machine code compatible with BVPU's architecture for loading and execution.

---

3. CPU Architecture

3.1 Registers

- **General Purpose Registers:** BVPU features four general-purpose registers: `X0`, `X1`, `X2`, and `X3`.
- **Usage in Assembly:** Registers are directly referenced by their names (e.g., `X0`, `X1`).

3.2 Supported Functions

The BVPU supports the following instructions:

- **Arithmetic Operations:** Addition (`ADD`) and subtraction (`SUB`), both with register-to-register and immediate formats.
- **Memory Operations:** Load (`LDR`) and store (`STR`) with both immediate and register offsets.

3.3 Instruction Formats

BVPU instructions are 16 bits long and come in two formats:

1. **Register Format:**

`[Opcode][Rm][xxx][Rn][Rd]`

   - `Opcode`: 7 bits (defines the operation type).
   - `Rm`: 2 bits (source register).
   - `Rn`: 2 bits (source register).
   - `Rd`: 2 bits (destination register).

2. **Immediate Format:**

`[Opcode][Imm5][Rn][Rd]`

   - `Opcode`: 7 bits (defines the operation type).
   - `Imm5`: 5 bits (immediate value).
   - `Rn`: 2 bits (source register).
   - `Rd`: 2 bits (destination register).

---

4. Instruction Set

4.1 Arithmetic Instructions

ADD (Addition)

- **Format:** `ADD Rd, Rn, Rm`
- **Binary Encoding:** `1100000 Rm xxx Rn Rd`

Example:

```
ADD X0, X1, X2
Binary: 1100000 10 000 01 00
```

ADD_I (Addition with Immediate)

- **Format:** `ADD Rd, Rn, Imm5`
- **Binary Encoding:** `0101000 Imm5 Rn Rd`

Example:

```
ADD X0, X1, 5
Binary: 0101000 00101 01 00
```

SUB (Subtraction)

- **Format:** `SUB Rd, Rn, Rm`
- **Binary Encoding:** `1110000 Rm xxx Rn Rd`

Example:

```
SUB X0, X1, X2
Binary: 1110000 10 000 01 00
```

SUB_I (Subtraction with Immediate)

- **Format:** `SUB Rd, Rn, Imm5`
- **Binary Encoding:** `0111000 Imm5 Rn Rd`

Example:

```
SUB X0, X1, 3
Binary: 0111000 00011 01 00
```

4.2 Memory Instructions

LDR (Load Register)

- **Format:** `LDR Rt, [Rn, Rm]`
- **Binary Encoding:** `1100011 Rm xxx Rn Rt`

Example:

```
LDR X0, [X1, X2]
Binary: 1100011 10 000 01 00
```

LDR_I (Load Immediate)

- **Format:** `LDR Rt, [Rn, Imm5]`
- **Binary Encoding:** `0101011 Imm5 Rn Rt`

Example:

```
LDR X0, [X1, 5]
Binary: 0101011 00101 01 00
```

STR (Store Register)

- **Format:** `STR Rt, [Rn, Rm]`
- **Binary Encoding:** `1000100 Rm xxx Rn Rt`

Example:

```
STR X0, [X1, X2]
Binary: 1000100 10 000 01 00
```

STR_I (Store Immediate)

- **Format:** `STR Rt, [Rn, Imm5]`
- **Binary Encoding:** `0001100 Imm5 Rn Rt`

Example:

```
STR X0, [X1, 4]
Binary: 0001100 00100 01 00
```

---

5. Implementation Details

Instruction Encoding

The choice to use a 7-bit opcode is due to the exclusion of branching operations in our CPU, along with the support of our 2 arithmetic operations and Load / Store Instructions. The 7 bits in the opcode directly correlate to key control signals that are required to complete these 4 operations. The immediate is 5 bits, allowing unsigned values from 0 - 31 to be represented. Due to the opcode being 7 bits, and registers being 2 bits each, we chose a 5 bit immediate to keep instructions encoded in a 16 bit format.

Sequential Datapath

The BVPU employs a single-cycle sequential datapath design, integrating an ALU, register file, control unit, instruction memory, and data memory, controlled via a single clock..