LC2K FSM Simulator

Rijun Cai
12348003

January 4, 2014

In this laboratory, an LC2K FSM simulator is implemented in C. It is based on the framework provided in the lecture. I finished the FSM part of the simulator. This document introduces some details of my implementation.

To keep the code neat and easy to read, some marcos are defined.

- `_S`

  ```
  #define _S(name) name: printState(&state, #name)
  ```

  Begin to define a state. With this marco, a state can be simply defined as follow.

  ```
  _S(fetch);
      bus = state.pc++;
      // The rest of fetch state
  ```

  The above code will be expanded as follow.

  ```
  fetch: printState(&state, "fetch");
      bus = state.pc++;
      // The rest of fetch state
  ```

- `_DISPATCH`

  ```
  #define _DISPATCH(code, label) if((state.instrReg & 0x1C00000) == (code << 22)) goto label
  ```

  Define a branch of the dispatching process. Used in `dispatch` state. With this marco, `dispatch` state can be defined as follow.

  ```
  _S(dispatch);
      _DISPATCH(0x0, add);
      _DISPATCH(0x1, nand);
      _DISPATCH(0x2, lw);
      _DISPATCH(0x3, sw);
      _DISPATCH(0x4, beq);
      _DISPATCH(0x5, jalr);
      _DISPATCH(0x6, halt);
      _DISPATCH(0x7, noop);
  ```

  The above code will be expanded as follow.

  ```
  dispatch: printState(&state, "dispatch");
      if((state.instrReg & 0x1C00000) == (0x0 << 22)) goto add;
      if((state.instrReg & 0x1C00000) == (0x1 << 22)) goto nand;
      if((state.instrReg & 0x1C00000) == (0x2 << 22)) goto lw;
      if((state.instrReg & 0x1C00000) == (0x3 << 22)) goto sw;
      if((state.instrReg & 0x1C00000) == (0x4 << 22)) goto beq;
      if((state.instrReg & 0x1C00000) == (0x5 << 22)) goto jalr;
      if((state.instrReg & 0x1C00000) == (0x6 << 22)) goto halt;
      if((state.instrReg & 0x1C00000) == (0x7 << 22)) goto noop;
  ```

- _REG1, _REG2, _DES_REG and _OFFSET

```
#define _REG1 state.reg[(state.instrReg >> 19) & 0x7]
#define _REG2 state.reg[(state.instrReg >> 16) & 0x7]
#define _DES_REG state.reg[state.instrReg & 0x7]
#define _OFFSET convertNum(state.instrReg & 0xffff)
```

Convenient access to registers and the offset field. Then reading and writing registers can be simplified as follow.

```
bus = _REG1;
_DES_REG = bus;
bus = _OFFSET;
```

The above code will be expanded as below.

```
bus = state.reg[(state.instrReg >> 19) & 0x7];
state.reg[state.instrReg & 0x7] = bus;
bus = convertNum(state.instrReg & 0xffff);
```